(4)

RADC-TR-88-188
Phase Technical Report
September 1988

THE BBN KNOWLEDGE ACQUISITION PROJECT

BBN Laboratories, Inc.

Sponsored by
Defense Advanced Research Projects Agency
ARPA Order No. 5290

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

THE BBN KNOWLEDGE ACQUISITION PROJECT

Glenn Abrett
Mark H. Burstein
John Gunshenan
Livia Polany

Contractor: BBN Laboratories, Inc.
Contract Number: F30602-85-C-0005
Effective Date of Contract: March 1985
Contract Expiration Date: January 1989
Program Code Number: 8E29
Short Title of Work: The BBN Knowledge Acquisition Project
Period of Work Covered: Mar 85 - Jul 87

Principal Investigator: Albert Stevens
Phone: (617) 873-3802

RADC Project Engineer: Sharon M. Walter
Phone: (315) 330-3564

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Sharon M. Walter, RADC (COES), Griffiss AFB NY 13441-5700 under Contract F30602-85-C-0005.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-188
6a. NAME OF PERFORMING ORGANIZATION BBN Laboratories, Inc. — 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)
6c. ADDRESS (City, State, and ZIP Code) 10 Moulton Street Cambridge MA 02238	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Defense Advanced Research Projects Agency — 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-85-C-0005
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd Arlington VA 22209	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 62301E — PROJECT NO. E290 — TASK NO 00 — WORK UNIT ACCESSION NO. 01
11. TITLE (Include Security Classification) THE BBN KNOWLEDGE ACQUISITION PROJECT	
12. PERSONAL AUTHOR(S) Glenn Abrett, Mark H. Burstein, John Gunshenan, Livia Polany	
13a. TYPE OF REPORT Final — 13b. TIME COVERED FROM Mar 85 TO Jul 87 — 14. DATE OF REPORT (Year, Month, Day) September 1988 — 15. PAGE COUNT 134	
16. SUPPLEMENTARY NOTATION N/A	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Knowledge Acquisition; Frame; KREME; Knowledge Base Editor; Knowledge Representation; Database Editor.
12	05		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

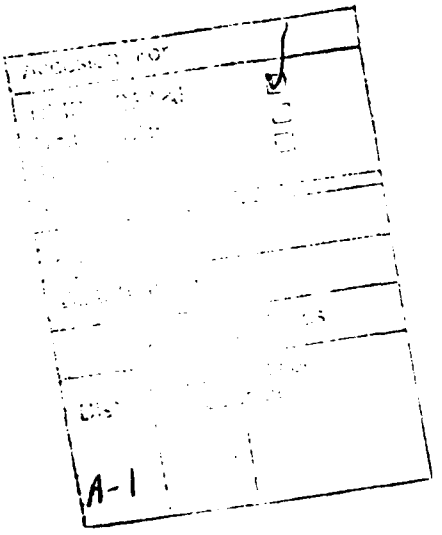
The goal of the Expert Assistant for Knowledge Acquisition project was to create a usable and extensible knowledge engineering environment that will be capable of handling very large knowledge bases, support experiments with knowledge engineering techniques and implement a useable system for knowledge acquisition and maintenance. During Phase One, KREME (Knowledge Representation Editing and Modeling Environment) was created. KREME is an extensible experimental environment for developing and editing large knowledge bases in a variety of representation styles. KREME is described in this report.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Sharon M. Walter	22b. TELEPHONE (Include Area Code) (315) 330-3564 — 22c. OFFICE SYMBOL RADC (COES)

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Table of Contents



List of Figures

1. Introduction

This is the Final Report for Phase One of the BBN Laboratories Knowledge Acquisition Project. This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Rome Air Development Center (RADC) under contract number F30602-85-C-0005.

The goal of this project was to create a useable and extensible knowledge engineering environment that will be capable of handling very large knowledge bases, support experiments with knowledge engineering techniques and implement a useable system for knowledge acquisition and maintenance. During Phase One of this project we have created the KREME Knowledge Representation Editing and Modeling Environment. KREME is an extensible experimental environment for developing and editing large knowledge bases in a variety of representation styles. It provides tools for effective viewing and browsing in each kind of representation, automatic consistency checking, macro-editing facilities to reduce the burdens of large scale knowledge base revision and some experimental automatic generalization and acquisition facilities.

Among the planned extensions to KREME are:

- The Procedure Editor
- A KEE Interface
- The Addition of Boolean Connectives to Slot Restrictions
- Extension of the Macro Editor

We are currently in the process of extending the value restriction language to permit more complex forms containing conjunctions, disjunctions and negations, based on the restriction language for KEE[tm1] frames [6]. This effort should result in an extended classifier, as well, capable of maintaining consistency among frames in the KEE class of frame languages.

During Phase Two we will also be developing experimental kinds of automatic knowledge acquisition: techniques for generating controlled acquisition dialogues, procedures to automatically transform previously acquired knowledge for use in new tasks, and techniques for learning by analogy and from examples.

The appendixes to this manual provide the detailed information needed by those who will be installing and using KREME at their sites. Appendix A provides instructions in loading KREME from tape. Appendix B is a User's Introduction to KREME. Appendix C presents a sample KREME session.

[1]KEE is a trademark of IntelliCorp.

2. Overview of the BBN Knowledge Acquisition Project

Our goal has been to develop an environment in which the problems of knowledge acquisition faced by every knowledge engineer attempting to build a large expert system are minimized. We believe both knowledge engineers and subject matter experts with some knowledge of basic knowledge representation techniques will find it easy to use KREME to acquire, edit, and view from multiple perspectives knowledge bases that are several times larger (i.e., 5-10,000 concepts) than those found in most current systems.

KREME attempts to deal with the inextricably related problems of knowledge representation and knowledge acquisition in a unified manner by organizing multiple representation languages and multiple knowledge editors inside of a coherent global environment. A key design goal for KREME was to build an environment in which existing knowledge representation languages, appropriate to diverse types of knowledge, could be integrated and organized as components of a coherent global representation system. The current KREME Knowledge Editor can be thought of as an extensible set of globally coherent operations that apply across a number of related knowledge representation editors, each tailored to a specific type of knowledge. Our approach has been to integrate existing frame and rule representation languages in an open ended architecture that allows the extension of each of these languages. In addition, we have provided for the incorporation of additional representation languages to handle additional types of knowledge.

Our approach to consistency maintenance has been to develop a *knowledge integration* subsystem that includes an *automatic frame classifier* and facilities for inter-language consistency maintenance. The frame classifier automatically maintains logical consistency among all of the frames or conceptual class definitions in a KREME frame base. In addition, it can discover implicit class relationships, since it will determine when one definition is logically subsumed by another, even when the knowledge engineer has not explicitly stated that relationship. The inter-language consistency maintenance facility checks for inconsistencies in references to frames in knowledge bases specified using other representation languages (e.g., rules, procedures).

A second important area of investigation in developing the KREME editing environment has been the attempt to provide facilities for large-scale revisions of a knowledge base. Our experience indicates that the development of an expert system inevitably requires such systematic revisions of the developed representation. This is often caused by the addition or redefinition of a task the system is to perform. These kinds of systematic changes to a knowledge base generally require painstaking piecemeal revision of each affected element, one at a time. Our initial approach has been to provide a *macro-editing* facility, in which the required editing operations can be demonstrated by example and applied to specified sets of knowledge structures automatically. A library of generic macro-editing operations for the most common and conceptually simple (though potentially difficult to describe) operations will be developed during Phase Two of the project.

Finally, we have begun to investigate techniques for *automatic generalization* of concepts defined in a knowledge base. We will briefly describe these experiments as well, in Section 8.

Underlying the entire KREME system is a strong notion of meta-level knowledge about knowledge representation and knowledge acquisition. The representation languages were implemented based on a careful decomposition of existing knowledge representation techniques and implemented as combinable objects using FLAVORS [7]. By organizing this meta-level knowledge base modularly, behavioral objects implementing such notions as inheritance and subsumption could be "mixed in" to a variety of representational subsystems making the incorporation of new representations and their editors reasonably straightforward. That is, each object in the meta-knowledge base encodes some aspect of a traditional representational technique, and is responsible for its own display, editing and internal forms.

3. The KREME Knowledge Representation Editing and Modeling Environment

3.1 Functional Description

The KREME family of knowledge editors currently consists of three major editor modules: a frame editor, a rule editor, and a procedure editor.[2] (See figure 3-1.) KREME also includes a large toolbox of editing techniques that are shared among the editor modules. This section will describe the global environment and toolbox, later sections will describe the individual editors. Sections 3.3 through 3.5 provide a discussion of the user interface. Readers who require more detail should consult Appendix B.

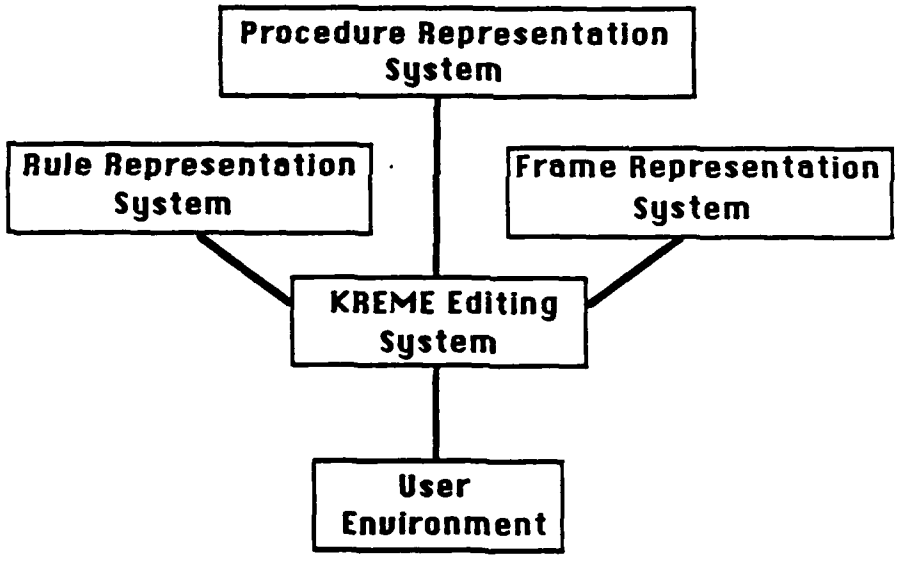


Figure 3-1: KREME: Functional Description

[2]The Procedure Editor is introduced briefly in this report. Full development of the Procedure Editor will occur in Phase Two.

3.2 Basic Editing Environment

Each type of representation included in the system has defined for it one or more editor *views*. A view is a collection of windows appearing together on the screen. Each window displays some aspect of the particular piece of knowledge being edited and/or a set of editing operations on it. When the user desires to enter or edit a specific piece of knowledge, the system opens the most appropriate view for the type of knowledge and the editing operation requested. Typically, any aspect of the knowledge being edited can be changed or viewed in more detail simply by pointing at it. This organization allows knowledge to be viewed by the user from multiple perspectives and at more than one level of detail.

The editor maintains a level of indirection between the knowledge being edited and the representation of that piece of knowledge in the knowledge base. This is accomplished by a mechanism like that of text editor buffers. Changes are always made to *editor definition objects* which are distinct from the corresponding objects in the actual knowledge base. The stack or list of the active definition objects is always visible to the user. The top item in this list is the definition currently being viewed and edited. The user is free to modify the current definition in any way without directly affecting the knowledge base. Only when the modified definition is to be placed into the knowledge base is a defining function appropriate to the type of knowledge (e.g., classification for concepts and roles), executed and the knowledge base modified.

Since the editor stack is always visible, it provides one convenient method for browsing. The user may point at any definition item currently in the stack. The object will then be displayed in the same editor view as when it was last edited.

A number of window subsystems or tools have been developed and incorporated into the KREME editor to make editing, viewing and browsing in knowledge bases easier and faster. They are described below.

3.3 The Grapher

KREME is equipped with a general graphing facility that rapidly draws lattices of nodes and links. Its main use is to provide a dynamically updated display of a concept or role and its place in the specialization or inheritance hierarchy. When editing a concept in the **Main Concept Editing View** or the **Big Concept Graph View**, or when editing a role, KREME automatically displays all of that object's abstractions and specializations. More abstract objects are displayed to the left of the current editor object, and more specialized objects to the right.

As shown in figure 3-2, the current editor object appears as a black node with white letters. All other objects appear as nodes with a white background. Objects that are defined as *primitive* are indicated by bold-edged boxes. Nodes that have been modified (edited but not reclassified) have a grey background.

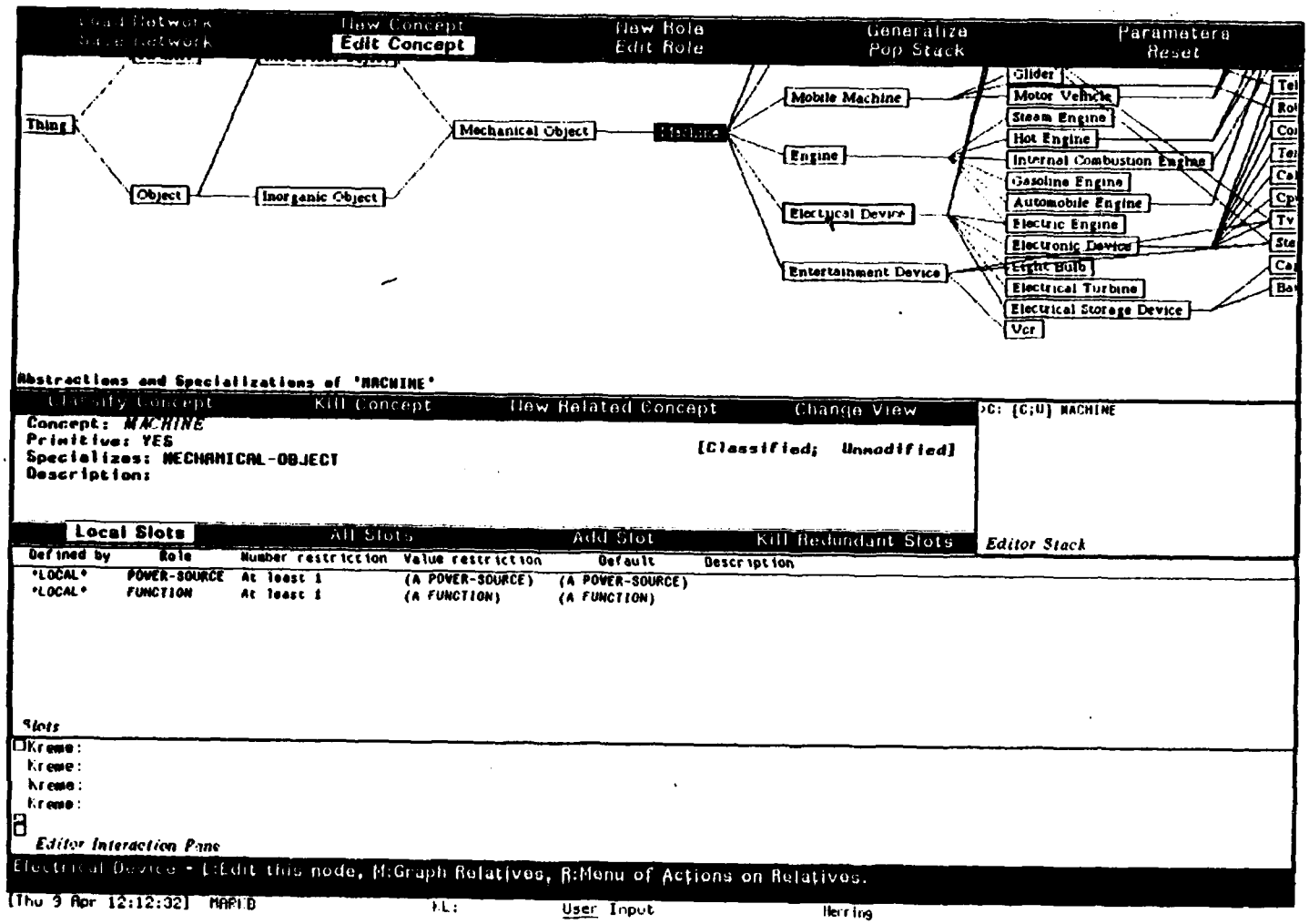


Figure 3-2: The Main Concept Editing View

3.3.1 Panning the Graph

The grapher can display a graph much larger than the window through which it is viewed. To see a part of the network that is off the screen, the user points with the mouse at some point on the graph window not containing a node, holds the left button down and drags the mouse. To *speed pan*, the user holds down the middle mouse button.

3.3.2 The Overview Graph

Clicking the right button once over an empty part of the graph window will make the **Graph Operations Menu** appear. If the user clicks **overview**, a miniature version of the full lattice will appear in a black region in the upper left corner of the graph window (as in figure 3-3). This overview shows a miniature version of the full network. The visible region of the graph is indicated by a white rectangle. If the user pans with the mouse over the main graph window, this white rectangle will follow the mouse movements. All of the mouse operations available on nodes in the main window will also work on nodes in this window. The name of the node being pointed at is indicated in the **mouse documentation window**. The overview window also can be used to pan the main graph window. The overview is turned off by bringing up the **Graph Operations Menu** and clicking the command **overview**.

3.3.3 The Graph Operations Menu

The other options in the **Graph Operations menu** shown in figure 3-4 are:

- **hardcopy** - Sends a copy of the full graph of the lattice to the printer.
- **style menu** - Allows the user to choose the font style and size of characters used for nodes on the graph. Smaller fonts are useful to see more of large networks at once.
- **find node** - Prompts for the name of an object on the graph, and centers that node on the graph window. It also draws a circle around the node so that the user can find it more easily. The circle disappears as soon as the graph is panned.
- **overview** - Switches the overview graph between visible and invisible.
- **orientation** - Switches the orientation of the graph. Normally, the lattice is drawn from left to right. This command will cause the graph to be redrawn from the top of the screen down, and vice versa.
- **speed pan** - This command pops up the speed panning box without having to hold down the mouse button. In this mode, clicking any mouse button will make it go away.
- **redraw graph** - Redraws the current graph.

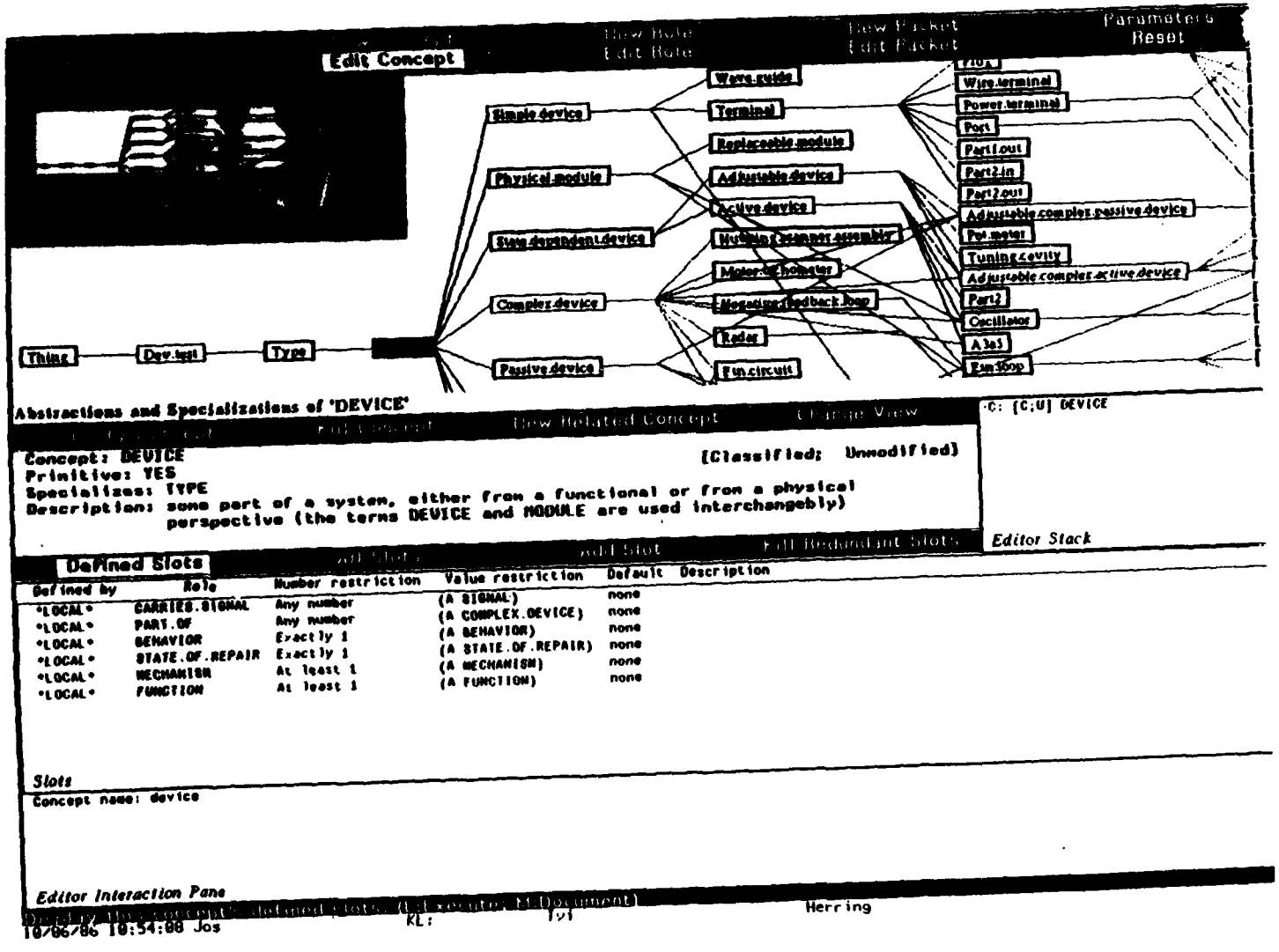


Figure 3-3: The Overview Graph

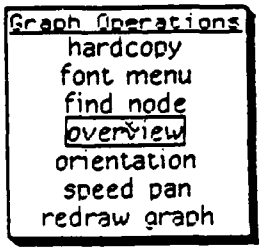


Figure 3-4: The Graph Operations Menu

3.3.4 The Graph Node Command Menu

Normally, the KREME Grapher displays only the abstractions and specializations of the current editor object, because KREME was designed to work with the very large lattices characteristic of *very large knowledge bases*. The Grapher provides a number of options to enable users to tailor the display to see more (or less) than KREME normally displays on such graphs.

Whenever the mouse is over a node on a graph, the **mouse documentation window** shows the name of the node, followed by:

```
L:Edit this node.  M:Graph Relatives  R:Menu of Editing Options
```

Clicking the left mouse button causes KREME to make the object pointed to the top editor stack item. This is an extremely convenient way of browsing through large concept networks quickly, and focusing on different portions of such a network. If, however, the user wishes to continue editing the concept that he is currently viewing, but see more (or less) of the network around that concept or some other concept *on the same graph*, he can use the **Graph Relatives Menu** found by clicking the middle mouse button over any graph node.

The **Graph Relatives Menu**, exposed by clicking the middle button over a node, contains the following commands:

- **Graph Parents** - causes all abstractions of the node clicked on to be added to the displayed graph.
- **Graph Children** - causes all specializations of the node clicked on to be added to the displayed graph.
- **Hide Children** - causes all specializations of the node clicked on to be removed from the graph, unless they are also children of some other node.

- **Hide Node and Children** - causes the node clicked on and its children to be removed from the graph.

3.3.5 Editing a Network from a Graph

Clicking the right button over a graph node causes yet another menu of options to be exposed, the Concept **Graph Edit Options Menu.**[3]

This menu contains the following options for concepts:

- **Show Definition** - This option causes the textual (LISP) form of the concept's definition to be displayed over the Graph Window.
- **Kill Concept** - This causes the concept pointed to to be removed from the knowledge base. It has the same effect as the **Kill Concept** command in the **local command menu window**, except that it works when the user is not currently editing the concept he wishes to kill.
- **Rename Concept** - This command prompts for a new name for the concept pointed to, and immediately replaces all references to that name with the new name throughout the knowledge base.
- **Delete Parent** - This command prompts for the name of a parent and then deletes that parent from the list of defined parents of the concept initially pointed to. It also switches KREME to editing the concept modified, so that it can then be reclassified.
- **Add Parent** - This command also prompts for a parent, adds the concept named to the list of defined parents of the concept, and switches to editing the modified concept.
- **Splice Out Parent** - This command prompts for a parent, and removes that parent from the list of defined parents of the concept, replacing it with *that* concept's parents. Again, the editor is switched to a view of the modified concept.

3.4 Editing in the State Window

The **state window** of the **Main Concept Editing View** displays basic information about the concept currently being edited. The top line displays the name of the concept, and any *synonyms* or alternate names for that concept. The name of the concept can be changed by clicking on the word **Concept:** and entering a new name.

The second line of the display shows whether the concept is defined as *primitive* or not, and whether the concept has been classified or modified since classification. Clicking on the word **Primitive:** causes the concept to be marked primitive if it was not, and vice versa.

The third line displays both the *direct and defined parents* of the concept, after the word **Specializes:**. *Defined parents* are concepts that the user specifies as abstractions of the concept. *Direct parents* are concepts that may or may not have been *defined* as parents of the current one, but have been determined by the classifier to

[3]On graphs of roles, the **Role Graph Edit Options Menu** appears, with essentially the same commands for roles, except as noted.

subsume the class denoted by this concept *and not have any specializations that also subsume this concept.* On the Concept Graph, the direct parents of a concept are the ones with direct links to it.

This Specializes: list should be read as follows: Concepts that are unmarked are both **defined parents** and **direct parents.** Concepts that are **defined parents** but not **direct parents** are prefixed by a "-". Concepts that are **direct parents** but not **defined parents** are prefixed by a "+". The user can easily add a parent to the set of defined parents of the concept.

3.5 Editing in the Table Edit Window

Normally, the **table edit window** in **Main Concept View** displays the set of **Local Slots** of the concept, that is, those slots which are defined locally by this concept and not inherited from above. The columns in the table are labeled **"Defined by", "Role", "Number Restriction", "Value Restriction", "Default", and "Description".**

Clicking (with the left mouse button) on the command **All Slots** in the **table edit command window** causes KREME to display both local and inherited slots. In this display, local slots are indicated by the word ***LOCAL*** in the **"Defined by"** column of the table. Slots inherited from a parent show the name of that parent. Slots formed by combining the value restrictions and/or number restrictions of several parents are indicated by the word ***CLASSIFIER***. When the table window is displaying all of the concept's slots, the user can return to viewing just the local ones by clicking the command **Local Slots.**

Whenever the Table Edit Window shows slots of the current concept, the user can edit those slots or add new ones. To change the slot name, value restriction, number restriction, default, or description of a slot, the user simply clicks the left mouse button over the thing to be changed, and will be prompted for a replacement. For all but number restrictions, the right button will pop up a menu that includes the commands: **Change** the part of the slot pointed to, **Show Definition** of the concept or role pointed to, **Edit Definition** of that concept or role, or pop up a **Graph** of its abstractions and specializations. When pointing to the slot name, in the column labeled **"Role"**, the user can also **Rename Role**, that is, change the name of the role, and all references to it in the knowledge base.

When the mouse is over a line in the slot table, and the entire line is encircled by a box, the right mouse button can be used to get a menu of **Delete Slot, Copy Slot** to another concept, and **Move Slot** to another concept. For the last two, KREME prompts for the name for the concept to move or copy the slot to.

3.5.1 Adding New Slots

Whenever the slots table window is visible, as in the Main Concept Editing View, the user can add new local slot definitions. A new slot is added to the defined slots of the concept with the Add Slot command. When this command is issued, the system prompts for a role name, a value restriction, a number restriction and a default form. Any of these items can be entered by typing or by pointing to the desired name or form if it is visible.

If a role or concept named in a role restriction or default does not exist, the system will offer to make one with the name given, and proceed to pop up the defining form for that object. When the user is finished filling out the form, he clicks Define, and KREME will continue to ask for the rest of the new slot's features.

When the user has finished adding and modifying the slots of a concept, he should always make the changes permanent with the Classify Concept command.

3.5.2 Modifying the Table Edit Window

The appearance of Table Edit Windows can be modified in several ways. The tables are scrollable in both the up-down and left-right directions.If the user does not wish to see some columns of the table, they can be selectively removed.

3.5.3 Changing the Contents of the Table Window

Since there is not enough room in the Main Concept Editing View to display all of a concepts defining features at one time, the contents of the Table Edit Window can be changed to display those other features. To do this, the user must use the mouse to find the table window contents menu. This menu is available wherever there is nothing else under the mouse while still inside the table window. the user will know he has found it because the mouse documentation window will show the words:

```
R: Change the contents of this table.
```

When the user clicks the right button, he will see the following menu options:

- Slots - Displays the table of this concept's slots, as described above.
- Inverse Restrictions - Displays a table, essentially like the slots table, but of all of the slots displayed are slots of other concepts that use the current concept as their *value restriction*. This table is useful when tracing references to a concept in other concepts. When this table is displayed, the table edit command window will be empty. Some of the editing options described for the slots table will not work here.
- Slot Equivalences - This table displays the slot equivalences of the current editor concept. This table has only three columns, "Defined by", "Path 1" and "Path 2". The two paths are designated as denoting the same object. Since slot equivalences can be inherited, their source is also indicated in the table, in the column "Defined by". When this table is visible, the table edit command window will show the commands Local Equivalences, All Equivalences, and Add Equivalence. The first two just

change which equivalences are displayed. The last prompts for two slot paths that should be made equivalent.

- **Disjoint Concepts** - This table is just a one column list of all of the concepts that are defined to be disjoint from the one currently being edited. When this table is visible, the **Table Edit Command Window** will display the commands **Add Disjoint Class**, **Local Disjoint Classes**, and **All Disjoint Classes**.

3.6 Files and Multiple Language Support

All definitions manipulated by the editor are read and stored in lisp-readable text files of defining forms. Since these files contain formatted lisp forms, they are user-readable, and can be edited offline using an ordinary text editor. In fact, KREME can as easily read files that were developed independently using a text editor or some other frame editor.

Files are read in using the LOAD command. A file can be loaded into a blank KREME knowledge base or can be loaded on top of an already existing knowledge base. This mechanism, which relies heavily on the the frame classifier to maintain consistency, enables KREME to organize information from multiple knowledge bases to create a single unified whole.

KREME currently reads and writes definitions in either its own frame language syntax or NIKL syntax. This flexibility has made it possible for KREME to be used regularly to examine and update a knowledge base of approximately 1000 roles and concepts for the IRUS/JANUS natural language interface that was built using NIKL. KREME can also read files of MSG (the frame language of the STEAMER [22] system) defining forms, providing access to the extensive STEAMER knowledge base of concepts and procedures. We are currently building an interface to files of KEE frame definitions.

This multiple language handling facility is a crucial feature of KREME. A library of input translation programs will enable a knowledge base builder using KREME to draw upon previously existing knowledge bases to create new knowledge bases.

4. The KREME Frame Editor

This section will describe the KREME knowledge editor for a frame representation language.

4.1 The KREME Frame Language

A number of frame languages have been developed in recent years to support AI systems [12, 2, 18, 9, 3, 6, 8]. These languages have all been well researched and extensively tested. For KRME, our most important criteria for a suitable frame representation language were that it:

1. Allowed multiple inheritance
2. Was a logically worked out mature language.
3. Had some mechanism for internal consistency checking.
4. Was built on a modular object oriented base so that the language could be decomposed in such a way as to make it easily extensible.

NIKL (the definitional or frame language component of KL-TWO) [9, 15, 21] seemed an ideal candidate. It is a fully worked out frame representation language that allows multiple inheritance, is reasonably expressive and, perhaps most importantly, was designed to work effectively with an automatic classification algorithm that could be easily adapted to provide a powerful mechanism for consistency checking and enforcement during knowledge base development. However, no object-oriented implementation of NIKL existed, and the NIKL classifier was not designed to allow *modification* and *reclassification* of previously defined concepts. A second frame language, known as MSG, had been built as part of BBN's STEAMER project and is object oriented in both of the above senses.

To develop KREME, we elected to reimplement NIKL as an object oriented language using MSG as a guide. The NIKL data structures were decomposed into a modular hierarchy of flavor definitions, and the KREME frame language was then built out of these flavors. This enabled us to incorporate the sophisticated instantiation mechanism of MSG with minimal effort. In the process, we were also able to implement a modular version of the NIKL classifier algorithm. This provided the kind of reclassification capability required for a knowledge editing environment and anticipated the extension of the classifier to deal with the richer semantics of languages like Intellicorp's KEE [6].

4.1.1 Frame Language Syntax

The remainder of this section will briefly describe the basic definitional syntax of the KREME Frame language. As this syntax closely resembles the formal syntax of NIKL interested readers are referred to [9] for more detail.

Following NIKL, a KREME frame is called a *concept*. Collections of concepts are organized into a rooted *inheritance* or *subsumption lattice* sometimes referred to as a *taxonomy* of concepts. A single distinguished concept, usually called THING, serves as the root or *most general concept* of the lattice. A concept has a *name*, a textual *description*, a *primitiveness* flag, a list of concepts that it *specializes* or is *subsumed by*, a list of *slots*, a list of *slot equivalences*, and a list of concepts that it is *disjoint from*.

The lists of slots, slot equivalences and disjoint concepts are collectively referred to as the *features* of a concept. If each concept can be thought of as defining a unique category, then features of the concept define the necessary conditions for inclusion in that category. If a concept is not marked as *primitive*, the features also constitute the complete set of sufficient conditions for inclusion in that category.[4] A concept inherits all features from those concepts above it in the lattice (those concepts that subsume it, and, thus, are more general) and may define additional features that serve to distinguish it from its parent or parents.

Slots (sometimes called role restrictions) consist of a role or slot name, a value restriction, a number restriction and an (optional) default form. The *value restriction* specifies the class of concepts allowed as values for that slot. As in NIKL, value restrictions usually specify a particular concept.

Slot Equivalences describe slots (and slots of slots) that *by definition* must always refer to the same entities.

The role name specified for each KREME slot refers to an object called a *role*. Roles in KREME, as in NIKL and several other frame languages like KRYPTON [3], and KnowledgeCraft [8], are actually distinct, first class objects that form their own distinct taxonomy, rooted at the most general possible role, usually called RELATION. Roles describe two place relations between concepts. A *role restriction* at a concept is thus a specification of the ways a given role can be used to relate that concept to other concepts.

[4]Concepts marked as *primitive* (sometimes referred to as *Natural Kinds*) have no complete set of sufficient conditions. For example, an ELEPHANT must, by necessity, be a MAMMAL, but without an exhaustive list of the attributes that distinguish it from other mammals, it must be represented as a primitive concept. The class of WHITE ELEPHANTs, on the other hand, might be completely described as a ELEPHANT, with slot COLOR restricted to WHITE.

4.2 Using the Frame Editor

The KREME frame editor has five views, the Main Concept Editing View, the Alternate Concept Editing View, the Big Graph View, and the Macro Structure Editor View. Roles, which are also part of the KREME Frame language, are edited with the Role Editing View. In this section, we will cover the details of the editing operations available in the first three of these views.

4.2.1 Editing in the Main Concept Editing View

Normally, when one creates a new concept or edits a concept for the first time, KREME makes that concept the top concept on the Editor Stack, and switches to display the Main Concept Editing View. There, KREME displays the concept as it exists at that time.

Figure 3-2 shows how the graph window immediately displays all of the abstractions and specializations of the concept being edited, the state window shows its name, whether it is primitive or not, its edit state (classified or not, modified or not), its parents, and a textual description. The table window simultaneously displays all of the concept's locally defined slots.

4.2.2 Frame Editing Operations

Space does not permit a full description of the functionality of the KREME frame editor so we will very briefly touch upon a few of its more important operations.

Making new concepts. The *New Concept* command in the global command menu initiates the definition of a new concept that is (1) fully specified by the user, (2) similar to some already defined concept, or (3) a specialization of one or several other defined concepts. When the initial form for the new concept has been specified the system creates a new concept definition for it and shows this new definition in the main concept view. The user is then free to add details (slots, equivalences, additional parents, etc.) to the new concept definition, classify it, or edit other concepts.

Adding and modifying slots. Whenever the window displaying slots is visible, slots can be added or modified. A new slot is added to the defined slots of the concept with the *Add Slot* command. Any portion of a slot's definition can be entered by typing or by pointing to a visible reference to the desired item. When a role or concept name that is not defined is specified, the system offers to make one with the name given.

Users may modify any locally defined slot or inherited slot. Slots shown in table windows are modified by pointing at the appropriate subform and then either typing in or pointing to a replacement form. Modifying an inherited slot causes the new definition to be locally defined.

Adding and Deleting parents. The system displays the classifier determined parents of a concept in two places. The concept graph displays them as part of the abstraction hierarchy of the concept, and the state pane indicates both the defined and direct or computed parents of the concept after the word "Specializes:". Since the classifier may have found that the concept being edited specializes some concepts more specific than those given as its defined parents, defined parents that are not direct parents are preceded by a "-", while classifier determined parents that were not defined parents are preceded by a "+".

Adding new defined parents to a concept's definition is done by clicking on the word "Specializes:" in the state window and typing a concept name or pointing to any visible concept. Parents can be deleted by clicking on their names in the list of parents displayed in the state window.

Changing names and killing concepts and roles. KREME allows the user to change the names of concepts and roles or to delete them completely. Name changing is accomplished simply by pointing at the concept or role's name in the state window and entering a new name. The *Kill* command splices a concept out of the taxonomy by connecting all of its children to all of its parents.

5. Large-Scale Revisions of Knowledge Bases

As knowledge bases grow larger, and the sets of tasks that intelligent systems are called upon to perform expands, system developers will need automatic methods for revising and reformulating accumulated knowledge bases. Toward this end, we feel that it is important to find ways of expressing *reformulations* of sets of frames and other representations and to begin developing facilities supporting the generation of new representations from old ones.

We are taking two different approaches to these problems. First, we have developed a macro facility for reformulations that can be expressed as sequences of standard, low-level editing operations. This facility allows users to use an example to define editing macros that can be applied to sets of frame definitions. Second, we are building a library of functions providing standard editing operations that cannot be defined simply as sequences of low level editing operations. Our main purpose in this project is to collect and categorize a number of different kinds of knowledge base reformulations. Our hope is that a large fraction of these operations can be conveniently described using the macro facility, as it is more accessible to an experimental user community than any set of "prepackaged" utilities, and can be more responsive to the, as yet, largely unknown special needs of that community

5.1 The Macro and Structure Editor

One of the views available when editing concepts in KREME is the *macro and structure editor*. This view (See figure 5-1.) provides display and editing facilities for concept definitions, based loosely on the kind of structure editor provided in many LISP environments. The view provides two windows for the display of stylized defining forms for concepts. The *current edit window* displays the definition of the currently edited concept (the top item on the editor stack). The *display window* is available for the display of any number of other concepts. Any concept which is visible in either window can be edited, and features can be copied from one concept to another by pointing Both windows are scrollable to view additional definitions as required.

There is a menu of commands for displaying and editing definitions that includes the commands **Add Structure**, **Change Structure**, **Delete Structure**, **Display Concept** and **Clear Display**. Arguments (if any) to these commands may be described by pointing or typing. Thus, to delete a slot, one simply clicks on **Delete Structure** and the display of the slot to be deleted. Adding a structure is done by clicking on **Add Structure**, the keyword of the feature class of the concept one wishes to add to (e.g., Slot:). The new slot itself may be copied from a displayed concept by pointing, or a new one may be entered from the keyboard. Changing (that is, replacing) a structure can be done by pointing in succession at the **Change Structure** command, the item to be replaced, and the thing to replace it with. In most cases, **Change Structure** can also be invoked simply by pointing at the structure to be replaced, without the menu command.

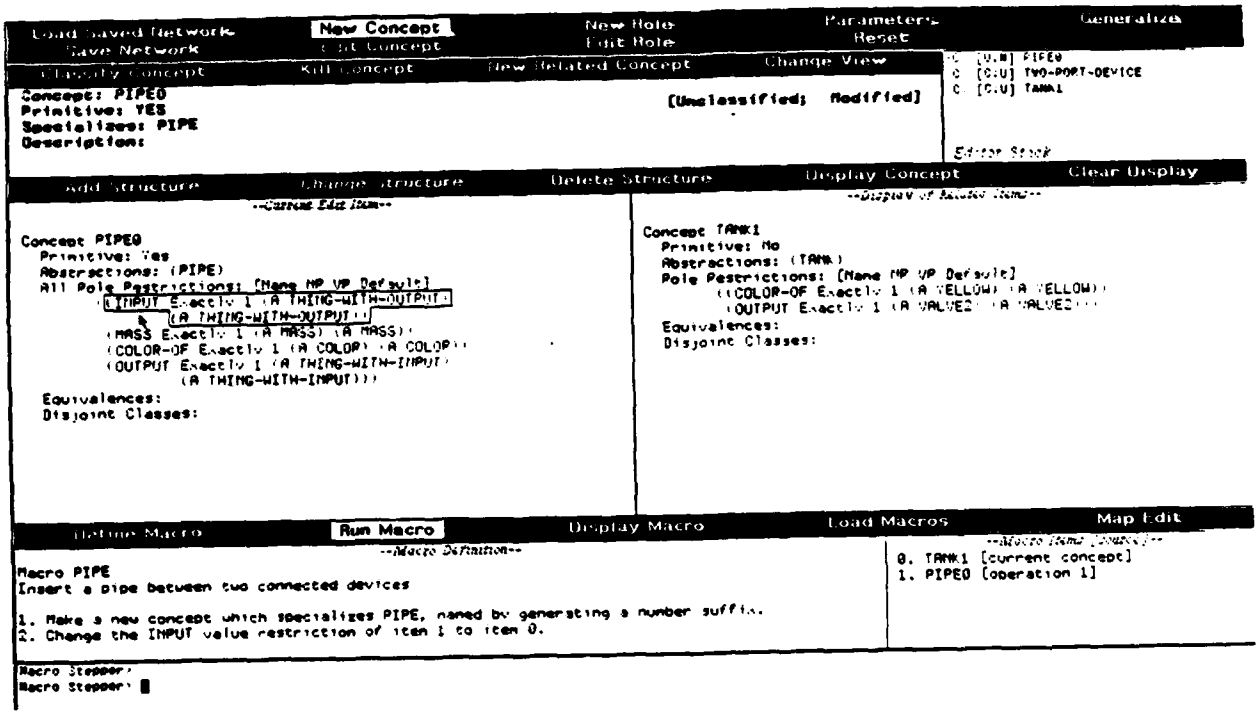


Figure 5-1: The Macro Structure Editor View

The last two commands in the structure view's main menu provide the means to change what is displayed in the display window. Pointing at **Display Structure** and then at any visible concept name places the definition of that concept in the display window. **Clear Display** removes all items from the display window. Individual concepts can be deleted from the display window by pointing at them and clicking. The **Edit Concept** command is used to change what is displayed in the current edit window. Editing a new concept moves the old edit concept to the bottom of the display window.

5.2 Developing Macro Editing Procedures

These operations, together with the globally available commands for defining new concepts and making specializations of old concepts essentially by copying their definitions, provide an extremely flexible environment in which to define and specify modifications of concepts with respect to other defined concepts. Virtually all knowledge editing operations can be done by a sequence of pointing steps using the current edit window and the display window. This style of editing is also used in the rule editor. The combination of editing features and mouse-based editor interaction style provides an extremely versatile environment for the description, by example, of a large class of editing macros.

In order to have macros, defined essentially by example, work on concepts other than those for which they were defined, the operations recorded cannot refer directly to the concepts or objects which were being edited when the macro was defined. This is handled by a kind of implicit variablization, where the objects named or pointed to are replaced by references to their relationship to the initially edited object. In most cases, these indirect references can be thought of as references to the *location* of the object in the structure editor's display windows. In fact, each new object that is displayed or edited in the course of defining a macro is placed on a stack called the *macro items list*, together with a pointer to the command that caused the item to be displayed. The utility of this form of reference will become clearer with an example.

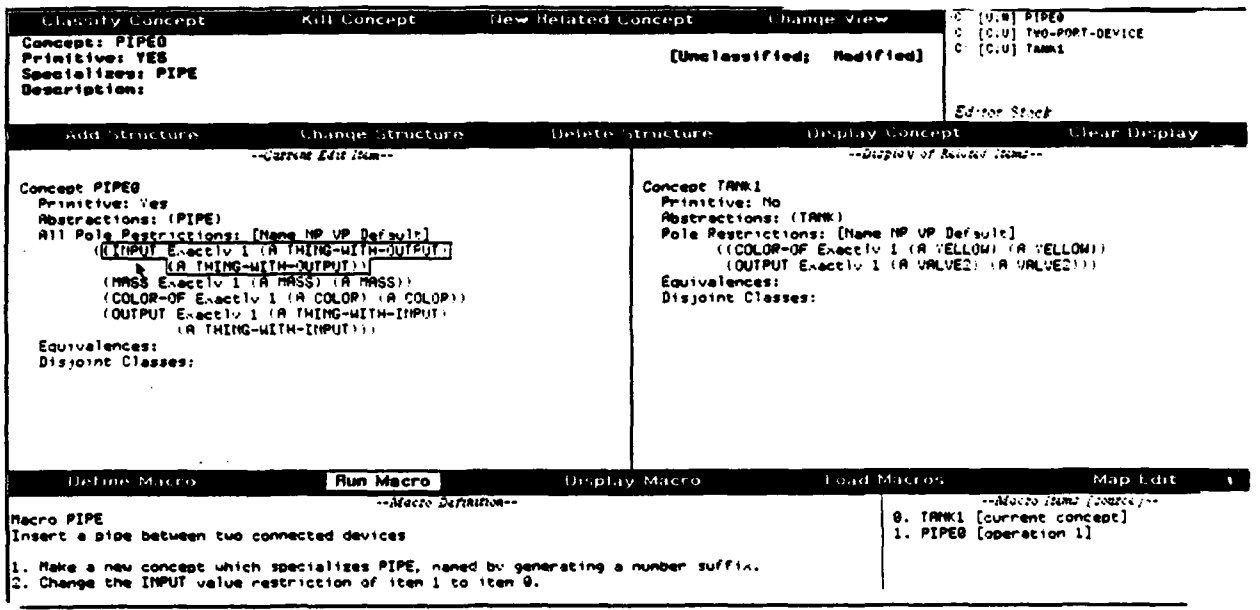
5.2.1 Macro Example: Adding Pipes Between Components

When the STEAMER [22] system was developed, a structural model of a steam plant was created to represent each component in the steam plant as a frame, with links to all functionally related components (e.g., inputs and outputs) represented as slots pointing at those other objects. So, for example, a tank holding water to be fed into a boiler tank through some pipe that was gated by a valve was represented as a frame with an OUTPUT slot whose value was a VALVE. The OUTPUT of that VALVE was a BOILER-TANK. The pipes through which the water was conveyed *were not represented* since they had no functional value in the simulation model. If it had become important to model the pipes, e.g. because they introduced friction or were susceptible to leaks or explosions, then the representational model that STEAMER relied on would have required *massive* revision. Each component object in the system would have needed editing to replace the objects in its INPUT and OUTPUT slots with new frames representing pipes that were in turn connected by their OUTPUT slots to the next component in the system.

One of our goals in developing the KREME macro editor was to be able to make such changes easily. While they are simple to describe, they normally require many tedious editing operations to a large number of concepts. Figure 5-2 shows a macro that can be applied to all objects in a system with INPUT and OUTPUT slots, in order to generate and insert PIPEs into those slots. The macro also sets the OUTPUTs of those PIPEs to be the concept that was the old value of the OUTPUT slot in the concept edited, and similarly redoes all INPUTs.

Figure 5-2 shows how the macro is defined, by editing a representation of a tank (TANK1) connected (by role OUTPUT) to a valve (VALVE2). The sequence of steps required, defined only using the mouse, is shown in figure 5-2, as they would appear in the *Macro Definition* window of the editor.

In Phase One, work on macro editing was only just begun. However, this technique already shows promise as a method for accomplishing restructurings of knowledge. We see our investigation of macro editing as only the first step in developing a knowledge reformulation facility that will make use of the higher level structure of the represented knowledge.



While Editing TANK1:
Click on **Define Macro**. (*Makes Macro Item 0 = TANK1*).

1. **Make a new concept which specializes PIPE.** (*Creates PIPE0 as item 1*).
2. **Change the INPUT value restriction of item 1 (*PIPE0*) to item 0 (*TANK1*).**
3. **Change the OUTPUT value restriction of item 1 (*PIPE0*) to the OUTPUT value restriction of item 0 (*OUTPUT of TANK1 = VALVE1*).**
4. **Classify the current edit concept (*Defines PIPE0*).**
5. **Change the OUTPUT value restriction of item 0 (= *VALVE1*) to item 1 (*PIPE0*).**
6. **Classify item 0 (*TANK1*).**
7. **Edit the OUTPUT value restriction of item 1 (*Creates item 2 = VALVE1*).**
8. **Change the INPUT value restriction of item 2 (*INPUT of VALVE1 = TANK1*) to item 1 (*PIPE0*).**
9. **Classify all items.**

Figure 5-2: Steps in PIPE Macro

6. Knowledge Integration and Consistency Maintenance

One of the most time consuming tasks in building large knowledge bases is maintaining internal consistency. Modification, addition or deletion of knowledge in one part of a knowledge base can have wide ranging consequences to both the meaning and structure of the knowledge stored in other parts of the knowledge base. A central component of the KREME system design was that it incorporate tools for consistency maintenance both within and across representation languages. These tools are collectively referred to as the *knowledge integrator*. When new knowledge is entered or existing knowledge modified it is the task of the knowledge integrator to propagate, throughout the knowledge base, the changes that this new or modified knowledge entails, and to report any inconsistencies that have been caused by the change.

In essence, the knowledge integrator takes each new or changed chunk of knowledge (e.g., a frame, role, rule or procedure) and determines, first, how the new definition fits into the knowledge base and, second, which other definitions depend on the current one for their meaning within the knowledge base. These dependencies are placed on an agenda which, in turn, causes them to go through essentially the same process.

The knowledge integration subsystem for frames is basically an extension of the *classification* algorithm developed for the NIKL representation language. The NIKL classifier correctly inserts *new* frames into their proper spot in a taxonomy, by finding the most specific set of concepts whose definitions *subsumed* the definition of the new concept. The KREME classifier was designed to additionally allow existing concepts and roles to be modified and and then *reclassified*, so that the effects of redefinitions are automatically propagated throughout the entire frame network. This was accomplished by redesigning the original NIKL classifier to take advantage of the meta-level descriptions of KREME Frames and implementing the new classifier using the dependency directed agenda mechanism of the overall knowledge integrator.

6.1 The Frame Classifier

The remainder of this section will give a brief description of the frame classification part of the knowledge integrator, which is the most completely developed portion of the system. For a formal description of the NIKL classifier algorithm see [15, 16]. For a more complete description of a somewhat simpler classifier for an editing environment, see [1].

The frame classifier works in essentially two stages, starting from a *concept or role definition*, as supplied by the editor or read from a file. The first stage, called *completion*, refers to the basic inheritance mechanism used by KREME Frames to install all inherited features of a concept or role in its internal description. The completion algorithm, when given a set of defined parents and a set of defined features for an object determines the full, logically entailed set of features of that object. The second stage is the actual classification or reclassification of a

role or concept. That is, the determination of the complete, most specific set of parents of the object in its respective subsumption hierarchy.

6.1.1 Completion

The completion algorithm is broken up into modular chunks that correspond to the decomposition of the frame language. There is a distinct component that deals with slot inheritance, another component that deals with disjoint class inheritance, a third that deals with slot equivalence inheritance and so on. This organization makes it quite straightforward to extend the language with new features that handle inheritance in different ways.

Figure 6-1 shows some of the complexities of slot inheritance. In 6-1A, the most specific *value restriction* for the slot LIMBS at 4-LIMBED-ANIMAL is inherited from one parent (ANIMAL) while the most specific *number restriction*, EXACTLY 4, is inherited from 4-LIMBED-THING. The completion algorithm determines that the restriction for the role LIMBS at the concept 4-LIMBED-ANIMAL must be EXACTLY 4 LIMBS.

Figure 6-1B shows one case for which the effective value restriction must logically be the conjunction of several concepts. Since ANIMAL-WITH-LEGS is both an ANIMAL, and a THING-WITH-LEGS, all of its LIMBS must be both ORGANIC-LIMBs and LEGs. If the concept ORGANIC-LEG, specializing both ORGANIC-LIMB and LEG, exists when ANIMAL-WITH-LEGS is being classified, the integrator will find it and make it the value restriction of the slot LEGS at ANIMAL-WITH-LEGS. If it does not exist, the integrator stops and asks if the user would like to define it (that is, define a concept that is both an ORGANIC-LIMB and a LEG).

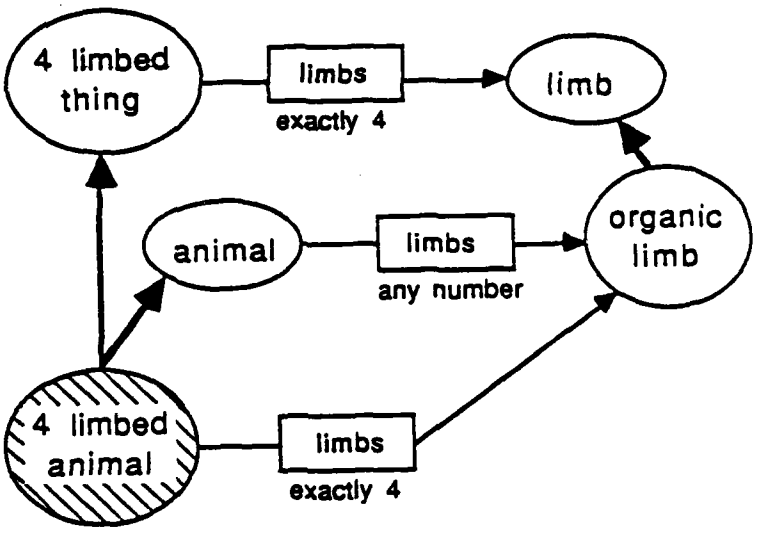
6.1.2 Classification

The second stage of the frame classification algorithm finds all of the *most specific subsumers* of the concept being defined of redefined. This is the actual *classification* stage, and is essentially a special-purpose tree walking algorithm.

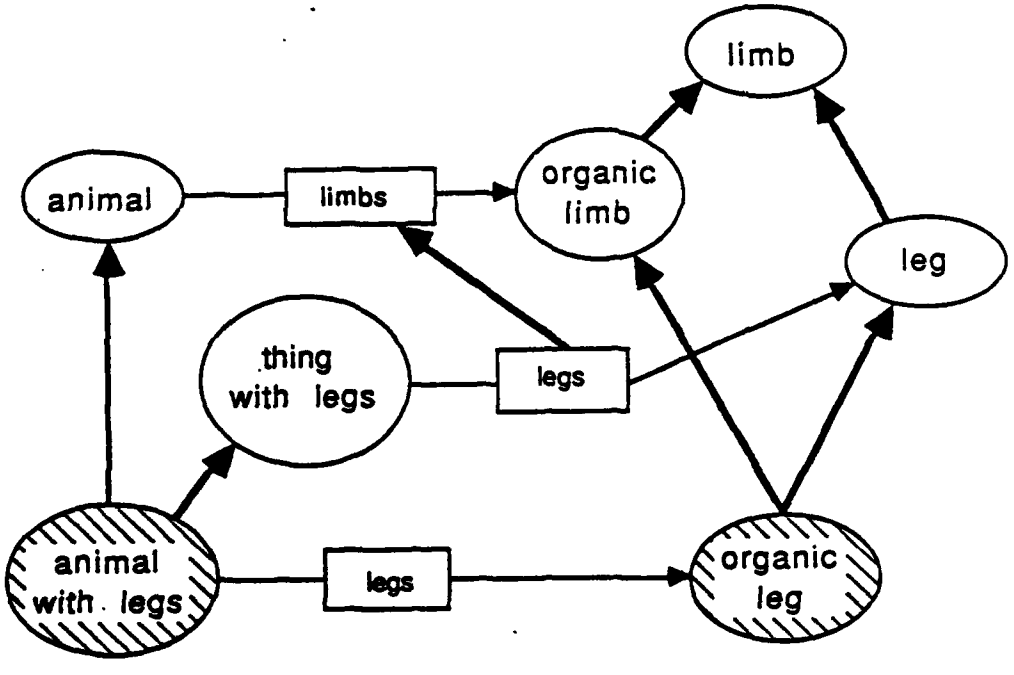
The basic classifier algorithm takes a completed definition (that is, a definition plus all its effective, inherited features) and determines that definition's single appropriate spot in the lattice of previously classified definitions. The result of a classification is a unique set of the most specific objects that subsume the definition and a unique set of the most general objects that are subsumed by the definition. When the classified definition is installed in the lattice all the concepts that subsume its features will be above it in the lattice and all the concepts that are subsumed by its features will be below it.

The classifier is built around a modularly constructed subsumption test that compares the completed sets of features of two objects. The object being classified is repeatedly compared to other, potentially related, objects in the lattice to see whether its completed definition subsumes or is subsumed by those other objects. For one definition to subsume the other, its full set of features must be a subset of the features of the other. As with

Figure 6-1: Two Examples of Slot Completion



Inheriting different number and value restrictions.



Conjoined Value Restrictions.

completion, subsumption testing is partitioned by feature type (i.e slot, disjoint-class etc). One object subsumes the other when all of its individual feature-type subsumption checks return EQUIVALENT or SUBSUMES, and there is at least one vote for SUBSUMES. The advantage of this kind of modular organization is extensibility. If a new feature type is added to the language one need only define a subsumption predicate for that feature, and objects having that feature will be appropriately classified.

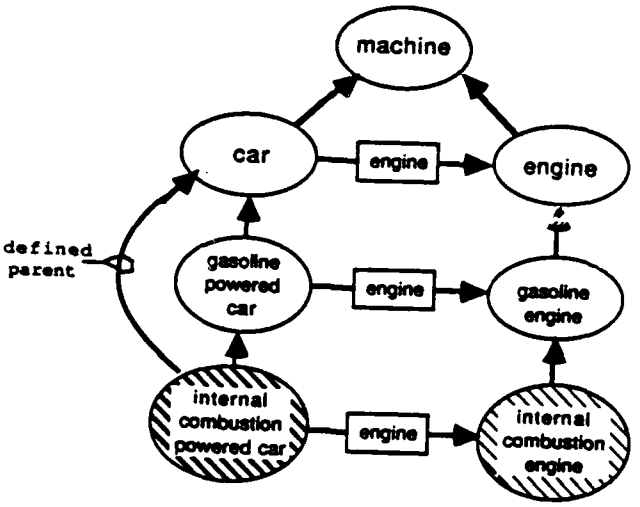
6.2 An Example of Reclassification

The power of frame reclassification in an editing environment can be illustrated with the following relatively simple example. Suppose a knowledge base developer had defined both GASOLINE-POWERED-CAR and INTERNAL-COMBUSTION-POWERED-CAR as specializations of CAR, but had inadvertently defined INTERNAL-COMBUSTION-ENGINE as a kind of GASOLINE-ENGINE. In this situation, the classifier would deduce that INTERNAL-COMBUSTION-POWERED-CAR must be a specialization of GASOLINE-POWERED-CAR, as shown in figure 6-2A, since the former restricted the role ENGINE to a subclass of the latter's restriction of the same role.

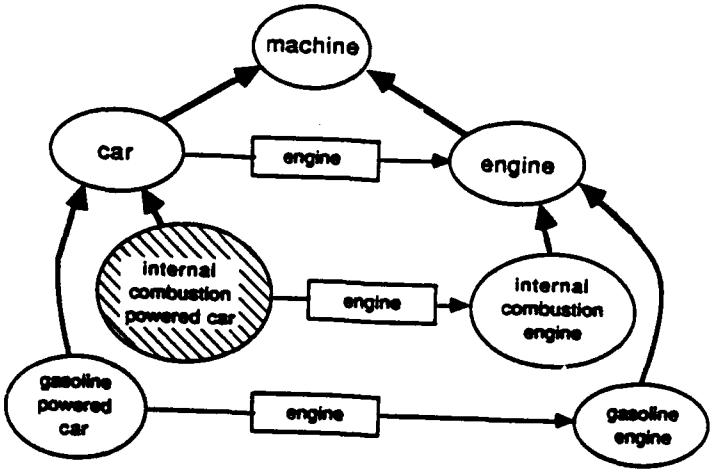
Redefining INTERNAL-COMBUSTION-ENGINE as a kind of ENGINE (rather than a GASOLINE-ENGINE), and then reclassifying, causes all of INTERNAL-COMBUSTION-ENGINE's dependents to also be reclassified, including INTERNAL-COMBUSTION-POWERED-CAR. Since GASOLINE-ENGINE no longer subsumes INTERNAL-COMBUSTION-ENGINE, the restrictions for GASOLINE-POWERED-CAR no longer subsume those of INTERNAL-COMBUSTION-POWERED-CAR, and the classifier therefore finds that GASOLINE-POWERED-CAR does not subsume INTERNAL-COMBUSTION-POWERED-CAR. This is shown in figure 6-2B.

The combination of inconsistency detection during the completion phase and the automatic propagation of classification changes that occurs during reclassification makes KREME a powerful and extremely useful tool for knowledge base development and refinement. Since the effects of reclassification are *immediately* made apparent to users via the dynamically updated graph of the subsumption lattice, they sometimes find that the definitions they have provided have some unanticipated logically entailed effects on their taxonomy. Sometimes these effects are surprising, although correct. Other times, they lead to changes and additions which make the knowledge base more complete and correct.

Figure 6-2: An Example of Reclassification



A. Before Reclassification



B. After Reclassification

6.3 Using the Knowledge Integrator to Partition and Merge Knowledge Bases

6.3.1 Load/Merge

Perhaps the single most important use for the Knowledge Integrator is to enable orderly merging of independently developed knowledge bases. The process of loading one knowledge base into another is made somewhat involved by the need to merge and/or split and rename concepts that have the same name in both networks.

There are a number of complex cases to deal with. The simplest case occurs when two definitions of the same concept have different but complementary attributes. The KREME merge logic simply forms the union of the attributes of both concepts and edits all pointers to either concept so that they point to the new, enriched concept. (See Figure 6-3.)

A somewhat more complex case occurs when slots shared by both concepts are given different restrictions. (See Figure 6-4.) The system chooses the most specific restriction for the slot.

If concepts with the same name have properties that make it impossible to merge them -- that is, the identical names really stand for different concepts in the two knowledge bases (6-5), then the system will inform the user of this fact and ask the user for a new name for one concept.

The user has some control over this entire process and can set switches which cause the system to always query when it finds two concepts with the same name, always merge concepts if it can, or never merge concepts, keeping the knowledge bases distinct.

6.4 Saving and Partitioning Knowledge Bases

Any time during the development of a knowledge base, the user can save the entire developing knowledge base to a disk file. This is a useful feature when developing small knowledge bases or working on a piece of a knowledge base that will later be merged into a larger whole.

Another useful facility is KREME's ability to partition a knowledge base along user-designated lines and save the partitions in distinct files. This is accomplished by allowing the user to designate a set of seed concepts. KREME will then create and save a partition of the entire knowledge base, based on the seeds. In an oversimplifed sense, the partition consists of the seeds, all specializations of the seeds, and all the concepts that the seeds either directly or indirectly depend on. This facility can be used to break up a single knowledge base into several overlapping subcomponents.

6.5 Using Merge and Partition to Build Larger Knowledge Bases

Taken together, the merge and partition facilities suggest an approach that we think will prove to be an extremely powerful paradigm in the building of very large, very complex knowledge bases. When a knowledge base grows to a size at which it becomes difficult to deal with in its entirety, the partition/save facility can be used to divide it into several overlapping logical subcomponents, each of which is a full scale, consistent knowledge base in its own right.

These multiple, smaller knowledge bases can be worked on independently of each other with full confidence that the loader/merger can put the independently built subcomponents together in an orderly, consistent fashion.

In Figure 6-5, there are two networks. The "ball" in Network 1 stands for a concept that is a kind of round object. In Network 2, the name "ball" stands for a kind of formal dance. These are different concepts with unmergeable properties. In both networks, *Event* and *Object* would be defined to be disjoint. In this case, the Merger would ask the user for a new name for one of the concepts and would keep them distinct.

Figure 6-3: Example One : Merging with Nonoverlapping Attributes

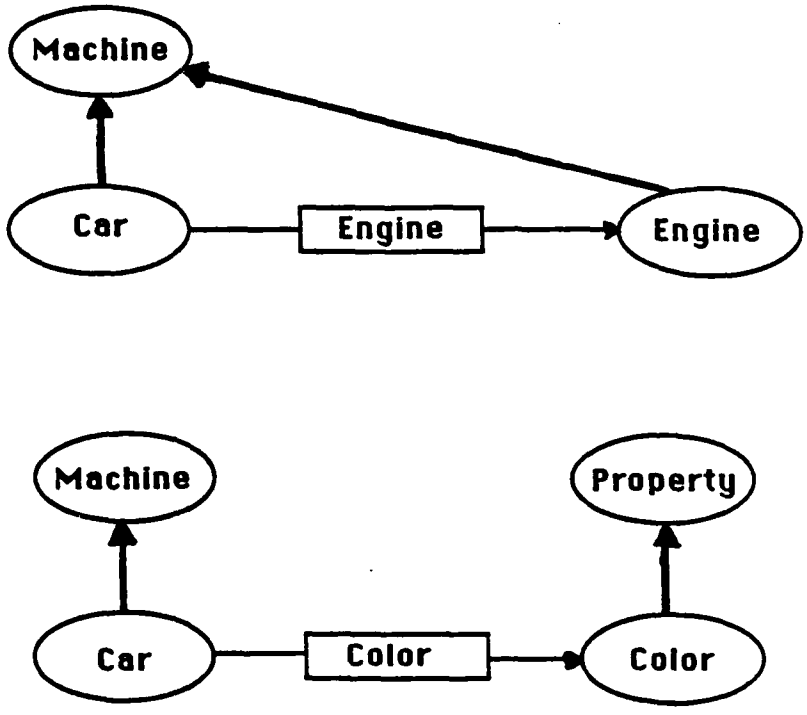


Figure 6-4: Example Two: Overlapping but Compatible Properties

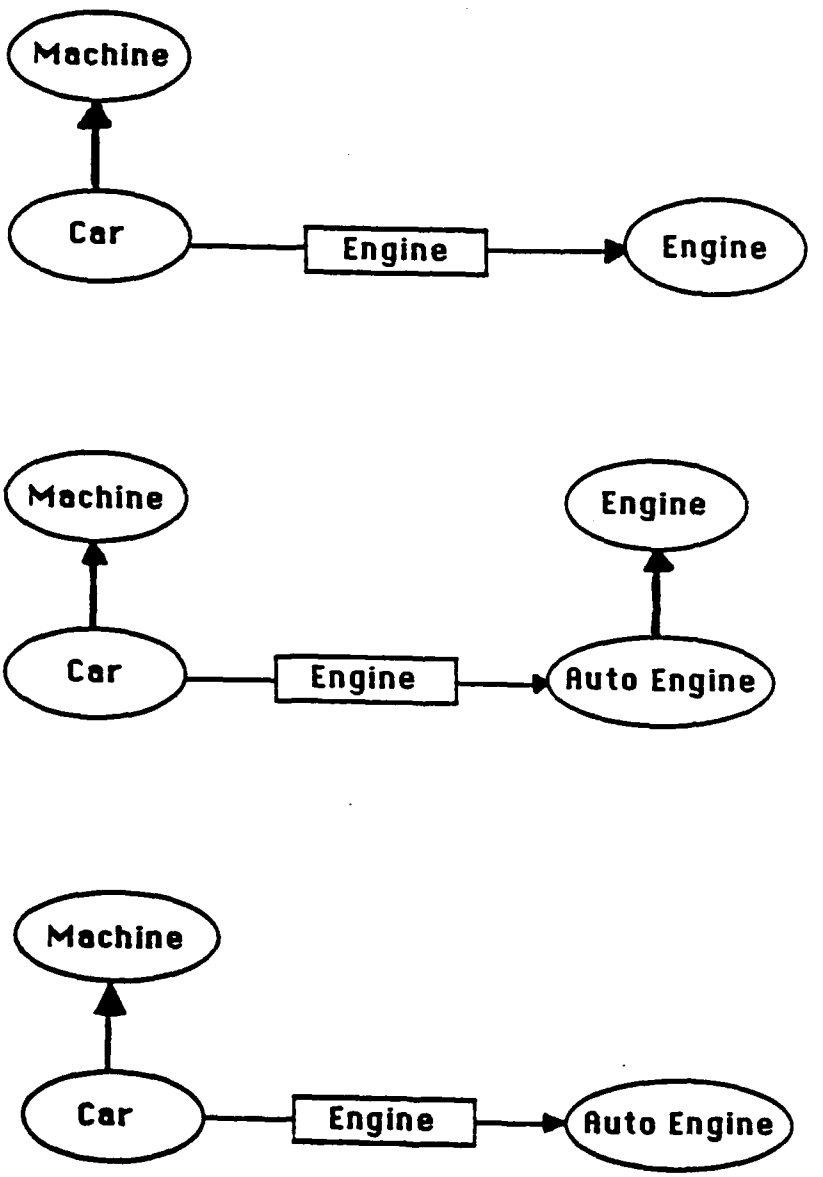
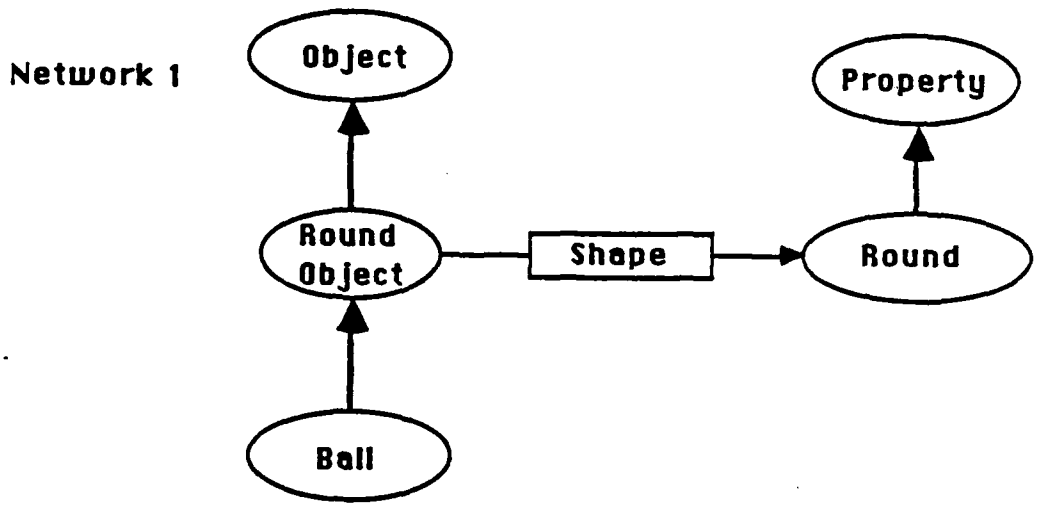
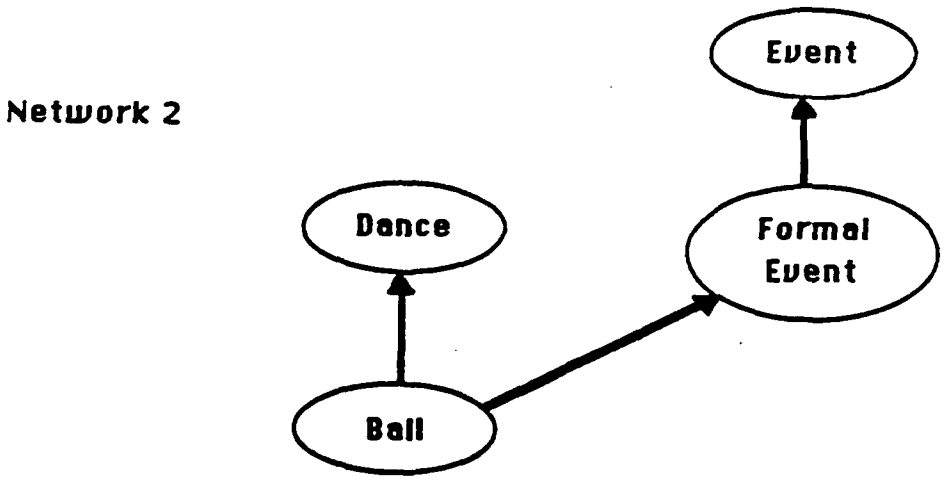


Figure 6-5: Example Three: Unmergeable Concepts





7. Editing Behavioral Knowledge

KREME embodies a set of mechanisms for representing and editing behavioral knowledge. One mechanism involves associating behaviors with frames. Since frames can also be associated with *flavors*, behaviors have been implemented so that they can be compiled into flavor methods.

A click of a mouse button and the *tabular features window* in the *main concept view* is turned into the toplevel behavior editor. All behaviors currently defined for the concept are shown. Each has a name and a type. There are three types of behaviors currently allowed; Rules, Procedures, and Methods. Existing behaviors can be edited or new ones defined. A modified form of the Symbolics[tm5] *flavor examiner* can be accessed to show various useful information about method combination and derivation.

Methods are simply flavor methods. Editing a method throws up a text editor window which can be interacted with in normal editing style or in structure editing style. Editing or inputting a new rule packet accesses the Rule Editor. Editing or inputting a new procedure accesses the Procedure Editor.

7.1 Editing Rules

The rule language used by KREME is a language called FLEX [17], based in large part on the LOOPS rule language. FLEX allows rules to be defined in *rule packets*, which organize sets of rules that are meant to be run together. In the KREME environment, rule packets can be attached to concepts, just as if they were functional methods. In addition, they may be inherited by more specialized concepts. FLEX incorporates a mechanism for dealing with uncertainty, based on EMYCIN [20]. The FLEX runtime environment also provides an elementary history and tracing mechanism, and an explanation system that produces pseudo-English explanations from rule traces. For efficiency, FLEX also provides a means for rule packets to be compiled as LISP code, and run without the rule interpreter present.

The KREME rule editor is built on top of the KREME structure editor. One defines and edits rules by specifying and filling out portions of rule *templates*. The user refines these templates either by using the mouse to copy parts of existing rules or by pointing at slots to be filled and typing in the desired values. Once a rule-set has been developed, the rule editor provides commands to run packets and debug them. It can also generate traces or rule histories paraphrased in pseudo-English. Mechanisms are also provided for deleting and reordering rules, and loading and saving them from files. The rule editor is shown in figure 7-1

The rule editor is also tied to the KREME's knowledge integration subsystem. At present, all references to

[5]Symbolics is a trademark of Symbolics, Inc.

slots of frames made in rules are checked for validity by the knowledge integrator. If invalid, the user is alerted and may switch, if necessary, to editing the associated frame. If the problem was simply that he/she named a non-existent slot, a valid one may be selected from a menu. In the near future, the knowledge integrator will also check such cross-references in the opposite direction, as when a slot referred to by some rules is deleted or changed in the frame editor.

KREME at present edits rules in the FLEX [17] rule language. In FLEX, rules come in *rule packets*, and the KREME Rule Editor edits an entire packet at one time. Rule packets provide a way to organize rules.

The forward chaining rule packets come in four varieties, indicating the type of control mechanism used for rule firing.

- **do-1-rule-packets** execute the first rule whose test succeeds.
- **do-all-rule-packets** execute all rules whose tests succeed.
- **while-1-rule-packets** repeatedly test all rules, firing one, until no tests succeed.
- **while-all-rule-packets** repeatedly fire all rules whose tests succeed, until none succeed.

Rule packets are connected to KREME frame systems or other data contexts by specifying an *access environment*. An access environment is an object that receives messages dealing with the accessing of values for references in the rules. It handles all messages to get or set the values of variables and their confidences.

7.2 The KREME Rule Editor

Rules are defined and edited by specifying and filling out portions of rule *templates*. To refine these templates either use the mouse to copy parts of existing rules or point at slots to be filled and type in the desired values.

There are also commands to run packets and debug them and to generate traces or rule histories paraphrased in pseudo-English, and delete rules and reorder rules, and load and save rules from files.

7.3 The Rule Editor View

Many of the windows in the **Rule Editor View** should be familiar by now. The complete list is as follows:

1. **Global Command Window** displays global commands that can be selected by the user. In this example, the user has used the mouse to select **Edit Packet**. The user's selection is highlighted.
2. **State Window** displays the name of the packet, the network it is associated with, and other useful information.
3. **Editor Stack Window** displays the names of the items recently edited and some information on their current state. Items in the editor stack window can be selected for editing with the mouse.

4. **Behavior Command Window** is a menu of commands that apply to Rules and Rule Packets. (*Behavior* is another term for rule packets, or functional methods on instances of concepts.)
5. **Current Edit Item Window** displays the item that has been selected for editing.
6. **Dispay Related Items Window** allows the user to view other rule packets and scroll through them. Rules and parts of rules can be copied from the Scroll Window into the Current Edit Item Window.
7. **Editor Interaction Window** displays screen prompts and user input. The user's edits are made in this window and then displayed in the Current Edit Item Window.
8. **Related Behaviors Window** displays an index of other rule packets that are related to the one currently being edited. With the mouse, the user can rapidly scroll through this index and select a related rule packet for viewing or editing.

To get into the Rule editor use the **New Packet** or **Edit Packet** command in the **global command window**.

Thereafter, the structure editor can be used in much the same way the Macro Structure Editor is used to edit concepts. The **Rule Structure Command Menu** contains the commands:

- **Define Behavior** is similar to **Classify Concept**. It makes the definition of the packet permanent, and allows it to be run or attached to a concept.
- **Similar Behavior** - Creates a packet with the same rules, etc. but gives it a new name, and presents it to be edited to make it different.
- **Kill Behavior** - Kills the definition of this packet.
- **Display Packet** - Displays the packet in the **Display of Related Items Window**.

When a whole rule packet is outlined, the user can choose to **Edit Packet** (L:), or (R:) choose from a menu of **Edit Packet**, **Edit Basis** or **Display Lisp Form**.

Other editing commands are found on the keywords and component pieces of packets and rules. For instance, clicking left on **Rule:** places a new (empty) rule in the packet, which can then be filled out by clicking on **IF** to add a new condition (conditions are treated as part of a conjunction) or **THEN** to add a new action. Clicking right gives a menu of **Add (Empty Rule)**, **Copy One Rule** from somewhere else into this packet, and **Copy Rule Set** which copies all of the rules from another packet.

Clicking over **Type:** gives the user a choice of the standard types of rule packets, described above.

Packet Classes: allows the user to specify a flavor to be mixed into the packet. **Arguments:** and **Return Variables:** each allow the user to add a new one (L:) or choose from a menu of **Add One**, **Add Several**, **Edit** and **Replace**.

When a whole rule is outlined, clicking left will be replace the rule with another rule that the user points at. Clicking right gives a menu of **Replace Rule**, **Edit Attributes** and **Delete Rule**.

Whenever expressions appear (after the word **Precondition:**, or as parts of conditions or actions), the user may **Replace** the expression (L:), or choosing from a menu (R:) of

- **Replace** the expression with another one.

- **Edit** the expression as text.
- **Delete** the expression.
- **Add Before** another expression (copied from somewhere by pointing).
- **Add After** another expression.
- **Exchange** two expressions positions.
- **Parenthesize** a set of expressions together.
- **Deparenthesize** an expression into pieces.
- **Evaluate** the expression in the current context.

7.4 Procedures in the KREME Environment

An obvious weakness of many knowledge representation languages is their inability to handle declaratively expressed knowledge about procedures as partially ordered sequences of actions, particularly if that knowledge is represented at multiple levels of abstraction. Although a number of systems have been developed that do various forms of planning, [5, 13, 14, 19], most have not encoded their plans in an entirely declarative or inspectable fashion. Certainly the current generation of expert system tools does not provide mechanisms geared to the description of this kind of knowledge. Although it is clear that much of an expert's knowledge about a domain is about procedures and their application, little work has been done on devising ways to capture that information directly.

The STEAMER project [22] began to address the issue of declarative representations for procedures in the course of developing a mechanism to teach valid steam plant operating procedures. The representation system developed for this task had to be directly accessible to the students who were the system's users, and it had to serve as a source of explanations when errors were made. STEAMER was able to describe these procedures, decompose them, show how they were related to similar procedures and, in general, deal with them at the "knowledge level" [10] rather than as pieces of programs or rule sets. Although the syntax of the language was quite primitive, with no provisions for branching or iteration; the mechanisms for procedural abstraction, specialization, and path or reference reformulation that formed the heart of the language seemed to form the kernel of an extremely useful representational facility.

The KREME representation language family includes a descendant of the STEAMER procedure language, built using KREME's library of knowledge representation primitives. Each KREME procedure has a *name*, a *description*, an *action* that the procedure is meant to accomplish, a list of *steps*, and a list of *ordering constraints* that determine the partial ordering of the steps. *Steps* have an *action* and an *object* which names the conceptual class of things that step acts upon. Procedures are attached to specific frames and can be "compiled" into flavor methods.

Each step in a procedure may either be a primitive action or another procedure. If the object of a step defines

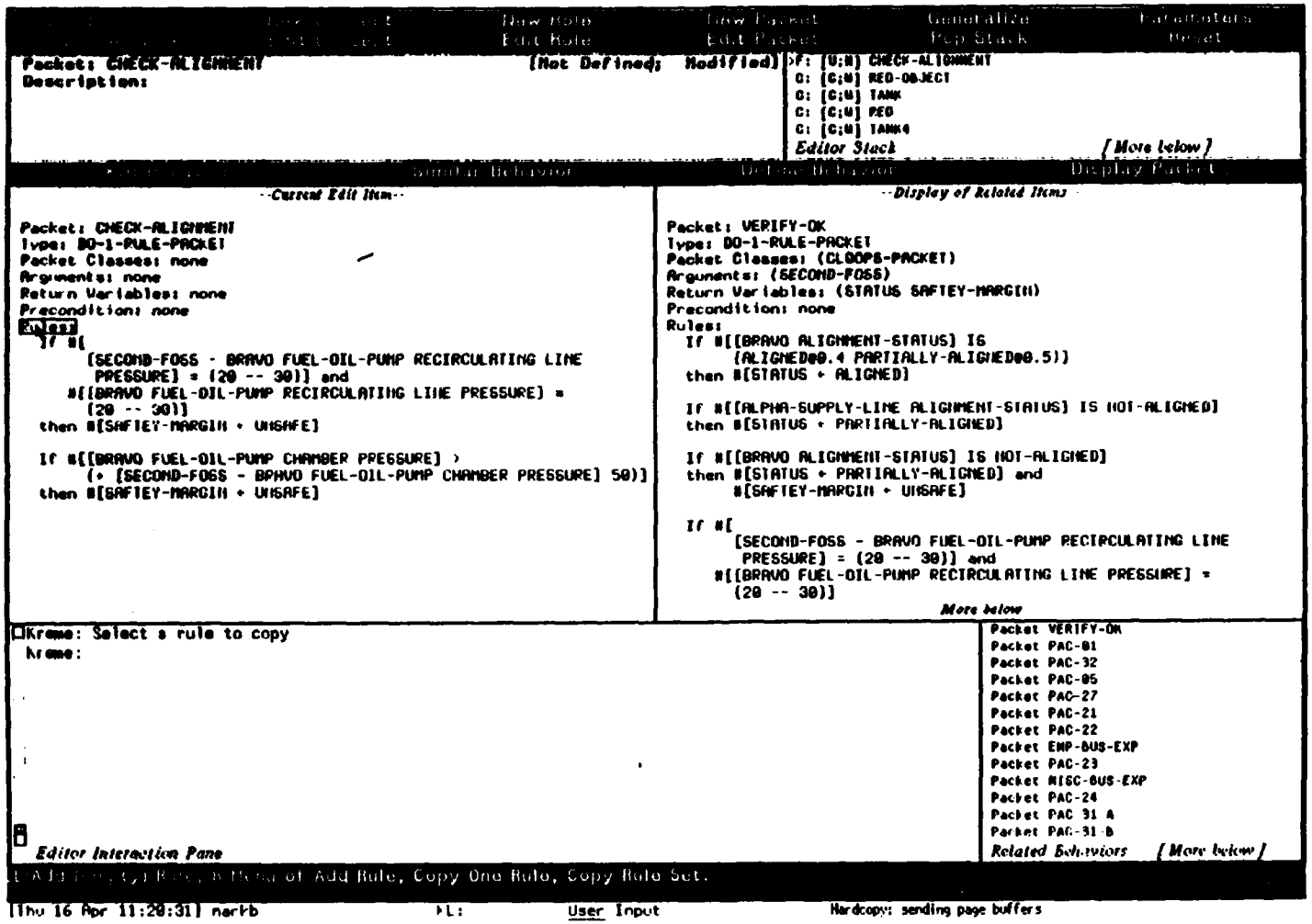


Figure 7-1: The KREME Rule Editor

a procedure for the action of that step then this procedure is said to be a sub-procedure of the enclosing procedure. For example, the ALIGN procedure attached to the concept SUCTION-LINE could have a step ALIGN <PUMP>. If the concept CENTRIFUGAL-PUMP, which is the object of this step for SUCTION-LINEs, defined a procedure for the action ALIGN, then the step ALIGN <PUMP> could be expanded into the steps of the procedure for aligning a centrifugal pump.

7.4.1 Procedural Abstraction and Structure Mapping

For knowledge acquisition purposes, it would be very useful if procedures were represented in an abstraction hierarchy like that for frames. In a strong sense, it seems difficult to define exactly what it means for one abstract procedure to subsume another. However, from an acquisition standpoint, much power can be gained by allowing abstract procedures to form templates upon which more specific procedures can be built, and eventually providing tools for automatic plan refinement like those found in NOAH [14]. For example, if you have some idea about how to grow plants in general, and you want to grow tomatoes, you will use your knowledge about growing plants in general as a starting point for learning about growing tomatoes. The final procedure for growing tomatoes will include some (presumably more detailed) versions of steps in the more general procedure, and may also include steps that are analogous to those used in growing other plants for which more detailed knowledge exists.[6]

The KREME Procedures editor has a mechanism for building templates of new procedures out of more abstract procedures. When a new procedure is being defined at a concept, the procedural abstraction function determines whether any of that concept's parents have a procedure for accomplishing the same action. If so, an initial procedure template is built by combining the steps and constraints of all the inherited, more abstract procedures. The paths (objects) of the steps are adjusted, using the concept's slot equivalences, to use "local" slot names, as much as possible. As yet this facility does not have the ability to do detailed reasoning with constraints on steps, as NOAH does. We expect to greatly expand this capability during Phase Two of the project.

[6] For a detailed discussion of related issues see Carbonell [4] on derivational analogical planning.

8. Knowledge Extension

One task faced by knowledge engineers is getting experts to express *generalizations* about their domains of expertise. While much of the detailed information about particular problems can be accessed and represented by looking at specific examples and problems, the expert's abstract classification of problem types and the abstract features he uses to recognize those problem types are less directly available. Experienced knowledge engineers are often able to discover and define useful generalizations which experts perceive as relevant to their own reasoning processes. The experts may then suggest improvements, related generalizations, or more abstract generalizations.

Our initial experiment in knowledge-base extension in Phase 1 has been the development of a *frame generalization* algorithm. Our current generalizer finds potentially useful generalizations by searching for sets of concept features that are shared by several unrelated concepts.

When the generalizer finds a set of at least k features shared by at least m concepts, where k and m are user-settable parameters, the system forms the most specific concept definition that would enclose all of the features but would still be more general than any concept in the set. Since our simple algorithm has no other external notion of "interestingness" it simply displays this potential new concept definition to the user. For example, given three concepts that are all ANIMALs and independently define the slot WINGS, the generalizer would suggest forming a specialization of ANIMAL with the slot WINGS, that these concepts would all specialize. If the user wanted to introduce this concept, he would respond by naming the new generalization (e.g., FLYING-ANIMAL), which would then be classified and integrated with the network. The features that are enclosed by this new, more general concept, are automatically removed from each of the more specific concepts being generalized.

9. Conclusion

The goal of the BBN Laboratories Knowledge Acquisition Project is to build a versatile experimental computer environment for developing and maintaining large knowledge bases. We are pursuing this goal along two complementary paths. First, we have constructed a flexible, extensible, Knowledge Representation, Editing and Modeling Environment in which different kinds of representations (initially frames, rules, and procedures) can be used. We are now using this environment to investigate acquisition strategies for a variety of types and combinations of knowledge representations. In building and equipping this "sandbox", we have been adapting and experimenting with techniques which we think will make editing, browsing, and consistency checking for each style of representation easier and more efficient, so that knowledge engineers and subject matter experts can work together to build significantly larger and more detailed knowledge bases than are presently practical.

The second aspect of our research plan is the development of more automatic tools for knowledge base reformulation and extension. An important part of this endeavor is the discovery, categorization and use of explicit knowledge about knowledge representations; methods for viewing different knowledge representations, techniques for describing knowledge base transformations and extrapolations, techniques for finding and suggesting useful generalizations in developing knowledge bases, semi-automatic procedures of eliciting knowledge from experts, and extensions of consistency checking techniques to provide a mechanism for generating candidate expansions of a knowledge base.

We are attempting to provide a laboratory for experimenting with new representation techniques and new tools for developing knowledge bases. If we are successful, many of the techniques developed in our laboratory will be adopted by the comprehensive knowledge acquisition and knowledge representation systems required to support the development and maintenance of future AI systems.

Appendix A
Loading KREME

A.1 Loading KREME from Cassette Tape

Each site can test KREME by loading KREME from tape according to the directions in this Appendix and then editing the sample networks provided on the tape. Once KREME has been loaded, Appendix B provides instructions on how to edit and create knowledge bases using KREME.

KREME requires a Symbolics machine with Genera-7.0 already installed and with at least 18000 blocks free in its FEP. If your machine has no tape drive, you will have to read the tape on another machine that does have one and then transmit the bands to your machine. (See section A.2) We will use the terms destination machine and tape drive machine to refer to these two machines. Note that you must have at least 18000 blocks free on the destination machine's FEP as well as having at least 18000 blocks free on the FEP of the machine with the tape drive.

A.1.1 Loading the FEP Files

There are four FEP Files on the tape. Your machine may already have inc-7-0G1-from-Genera-7-0.load. If so, do not create a FEP file for that file and do not load it from the tape.

Log in to the machine and create three (or four) FEP files in the following way:

```
Create FEP File inc-7-0G1-from-Genera-7-0.load  1290
Create FEP File inc-BBN-from-inc-genera-7-0G1.load  5600
Create FEP File Kreme-from-Boot7.load  9580
Create FEP File Kreme.boot  1
```

Log out and halt the machine.

Put the FEP Files tape in the tape drive .

Type the following to the FEP:

```
scan V127-disk
```

(This teaches the FEP about disk restore.) Then type

```
disk restore
```

The machine will then ask if you've done Set Disk Type. Answer Y[7] The machine then asks if you want to restore the FEP files on the tape. In each case answer Y and press carriage return. (If you already have the first band on your system, answer N for that band.) In each case, the system will then prompt

[7] If the disk is new and has not been initialized, see your local system wizard.

```
file to restore?
```

Accept the default file name by pressing carriage return.

The machine displays numbers as it reads from the tape. The machine then asks about the other files in turn. Each time, answer Y to restore the file and then press carriage return to accept the default file name.

A.1.2 Editing the FEP Files

Now you must edit the file Kreme.boot to set the CHAOS address correctly. To do this, boot the machine (using a boot file other than Kreme.boot) and edit Kreme.boot. Change the line containing the CHAOS address to set it to the address of the destination machine. You can get the correct CHAOS address for the destination machine from the system manager or by looking at the address in another .boot file on the destination machine.

You must also edit the Load Microcode line in Kreme.boot so that it contains the number of the microcode version on the host. To determine that number, ask the system manager or look at a .boot file that boots a 7.0 world.

Now log out and halt the machine.

A.1.3 Booting KREME

Type the following to the FEP:

```
Boot Kreme.boot
```

Because the band is being booted at a site other than the site at which it was built, the machine will ask you if the site is still BBN. Answer NO and the machine will name itself DIS-LOCAL-HOST.

If the machine has identity problems (It thinks it is still at BBN.), the simplest way to deal with them is to unplug the ethernet before booting Kreme.boot. See your local system wizard if you want a more elegant solution.

Once the boot is complete, you'll have a KREME window with the

```
KREME:
```

prompt. Now get to a Lisp Listener via

```
<select>L
```

Then log in with the command

```
(si:login-to-sys-host)
```

Logging in in this way avoids interacting with the BBN system accounting software. Then load the carry-tape with the command

```
(tape:carry-load)
```

The carry-tape contains two sample KREME networks, mech-net.lisp and org-net.lisp. You will have to choose a place on your machine to store these files.

You are now ready to use KREME with the help of *KREME: A User's Introduction*. Try loading a sample network from one of the files you read off the carry-tape.

A.2 For Machines with No Tape Drive

First, load the FEP Files from the tape onto the tape drive machine by following the instructions in section A.1.1. Then boot that machine, using a boot file other than Kreme.boot. Then transmit the FEP Files to the destination machine by typing the following to a Lisp Listener: (Answer Y when the system asks if you really want to.)

```
(si:transmit-band "fep0:>inc-7-0G1-from-genera-7-0.load"
                                        destination-machine)

(si:transmit-band "fep0:>inc-bbn-from-inc-genera-7-0.load"
                                        destination-machine)

(si:transmit-band "fep0:>Kreme-from-boot7.load"
                                        destination-machine)

Copy File fep0:>Kreme.boot  destination-machine|fep0:>Kreme.boot
```

You are now finished using the machine with the tape drive. You may delete the KREME files on that machine before going to the destination host.

Now continue with the instructions in section A.1.2.

Appendix B
A User's Introduction

Abstract

This appendix provides an introduction and preliminary user's manual for KREME, BBN's Knowledge Representation, Editing and Modeling Environment. KREME has been engineered to enable users to represent much of their knowledge about a domain while minimizing the classic problems of knowledge acquisition when building large expert systems. The manual documents KREME's component editors for two distinct representation languages; KREME Frames and KREME Rules. In order to maintain internal consistency in a Frame Knowledge Base, a problem which becomes increasingly more complex as taxonomies get larger, KREME provides a classifier to automatically check subsumption relations between frames. The KREME editing environment provides a macro-editing facility, for large-scale revisions of portions of a knowledge base. The macro editor allows sets of operations to be performed repeatedly over portions of a knowledge base. The required editing operations can be demonstrated by example and applied to specified sets of knowledge structure automatically. The KREME Rule Editor provides full support for important rule editing operations.

B.0.1 Introduction

This report provides a user's manual for KREME, BBN's Knowledge Representation, Editing and Modeling Environment. KREME was designed to facilitate the process of developing and editing representations of knowledge about a domain, while minimizing the classic problems of knowledge acquisition that come up during the development of large expert systems. Knowledge engineers and subject matter experts with some knowledge of basic knowledge representation techniques will find it easy to use KREME to acquire, edit, and view from multiple perspectives knowledge bases that are several times larger than those found in most current systems.

B.0.2 Introducing KREME

KREME is perhaps best thought of as a family of related editors for different styles of knowledge representations. The current version of KREME provides, within a uniform environment, a number of special purpose editing facilities that permit knowledge to be represented and viewed in a variety of formalisms appropriate to its use, rather than forcing all knowledge to be represented in a single, unitary formalism. In addition to a general editing environment, KREME provides tools to do the kinds of validation and consistency checking so essential during the development or modification of knowledge bases. As the size of knowledge bases grows, and more people become involved in their development, this aspect of knowledge acquisition becomes increasingly important. In the hybrid or multi-formalism representational systems that are becoming prevalent [11, 3, 21], techniques must be provided for consistency checking not only within a single representational system, but between related systems.

At present, KREME contains individual editors for three distinct representation languages; one for frames and

one for rules, and one for representations of ordered sequences of steps in operating procedures[8].

Frames, (also known as Concepts), are the primary way of expressing *declarative* knowledge about classes or *kinds* of things, both physical objects and abstract concepts or terms. Each concept or frame defined by a user is meant to stand for a class of things of a particular kind. Frames have, as part of their definitions, a set of *slots*, denoting the different *relationships* that things of that kind may, in general, have with other objects or concepts. The names of slots refer to *roles*, which are independently defined.

Rules are the primary way of expressing knowledge about *inferential procedures*. The basic form of a rule is IF *(condition)* THEN *(action)*. Rules are normally clumped together in *rule packets*. A packet is a set of rules whose conditions are checked when trying to make a specific decision about something. In KREME, one edits a whole packet at one time, rather than individual rules.

KREME has a number of useful features that enable it to make inferences about the knowledge it is given. KREME Frames are represented in a hierarchical network of more and more abstract classes. Any concept below another concept in the network *inherits* certain attributes from given information about concept(s) above it; these superordinate concepts are called *parents* of the subordinate concept. KREME's graphic components help you to construct networks quickly and easily. Editing features facilitate adding new concepts by taking advantage of similarities among to-be-added concepts and existing ones.

One of the most time consuming tasks in building knowledge bases is maintaining internal consistency. Adding, deleting and modifying slots and parents in a frame taxonomy may affect the subsumption (parent/child) relations between frames and, perhaps more importantly, may change sets of properties inherited by more specific frames. The possible consequences of a change in one part of a network grows rapidly as taxonomies get larger. Consequently, the size and complexity of knowledge bases is limited by the extent to which automatic means are provided for consistency checking.

The KREME *classifier* helps the user maintain consistency between the definitions of all concepts defined in a KREME Frame knowledge base. It is invoked whenever a concept or role is defined or redefined. The classifier first gathers all of the features to be inherited by a concept, and then determines exactly where the concept should be placed in the specialization hierarchy, by deducing what its most specific parents and least specific children should be. The classifier makes sure that the parents of a concept include not only those concepts that the user has specified directly, but all concepts that describe more general classes that *logically* include the given concept as a subclass.

The KREME editing environment provides facilities for large-scale revisions of portions of a frame knowledge base, in the form of a *macro-editing* facility. This facility provides editing operations that can be built up into little "scripts", and then applied repeatedly to many definitions. These "scripts" or *macros* are demonstrated once, by example, and then can be used over and over again.

---

[8]The procedures language, based on work done for the STEAMER ICAI system [22], is still under development, and will not be discussed further in this manual

The development of the macro editor was inspired by our experiences developing other expert systems. We found that over the life cycle of such systems, they inevitably require systematic, large scale revisions of portions of the developed representations. This kind of large-scale revision is caused by the addition or redefinition of a task the system is to perform. Previous to KREME, such systematic changes to a knowledge base have only been possible by painstaking piecemeal revision of each affected element.

B.0.3 Overview of this manual

Our general strategy in this manual will be to provide a brief introduction to important aspects of the system being discussed at the beginning of a section and to provide detailed information about the KREME facilities and procedures for using those facilities later in the section.

We begin in section B.1 with an overview of the KREME environment, providing details of the basic features of the knowledge editing environment that are common to both the KREME frame and rule editors.

Using KREME to edit Frame knowledge bases is discussed extensively in sections 3-5. Following a brief discussion of KREME frames, section B.2 details of the KREME system for representing knowledge in frames and editing those frames are presented. Section B.3 provides a brief discussion of the frame classifier and interactions with it. The macro editor is described in section B.4.

Section B.5 gives a brief description of the KREME mechanism for finding generalizations in KREME Frame Knowledge Bases. The KREME rule editor, and its relationship to the frame editor is described in section B.6. In Appendix C, we have provided an example of how to create new concepts using the KREME frame editor.

B.1 The Knowledge Editor

In this section, we describe the overall design of the KREME environment, and give details of the features of KREME that are universal to all of its component editors.

B.1.1 Windows and Views

We first present some of the basic design features of KREME, and how it appears to the user. This section will deal with the appearance of the screen when using KREME.

At any given time while using KREME, you will be looking at one of a number of *views* into a developing knowledge base. Each view is a collection of rectangular *windows* that together fill up the Symbolics screen. We have tried to select and arrange the windows in each view so that you can edit a particular kind of knowledge representation effectively and conveniently.

B.1.1.1 Views

A *view* is a particular editing perspective on some aspect of a knowledge base or representation of an object. In KREME, each view is a set of windows appearing simultaneously on the Symbolics screen. Figure B-1 shows the six views currently available. They are:

1. **The Top Level View** is seen when you first enter KREME, and whenever you are loading a previously saved knowledge base.
2. **The Main Concept Editing View** is the standard view for editing individual concepts.
3. **The Alternate Concept Editing View** contains windows available in the Main Concept Editing View (Slots, Inverse Restrictions, Equivalences, Disjoint Concepts), but displays the tables of concept features that are not normally visible all at one time. It does not show the Concept Graph.
4. **The Big Concept Graph View** uses most of the screen to show the Concept Graph, and does not show any tables of concept features.
5. **The Macro Structure Editor View** provides an alternative method for viewing and editing individual concepts. More importantly, it provides a convenient means of viewing a number of concepts at one time, *copying* features from one concept to another, and defining and running *knowledge editing macros*. This editing system is described in detail in section B.4.
6. **The Role Editing View** is used to edit roles, the relationships that name slots. It is much like the Main Concept Editing View.
7. **The Rule Editor View**, which operates much like the macro structure editor, is used for editing the rules in rule packets.

As you can see, many of the windows that appear in these views are "shared" by several different views. This is part of the basic design of KREME, to provide a uniform style of interaction while focusing on what is needed for editing each type of representation. Next we discuss what these windows are, and what they are for.

B.1.1.2 Component Windows

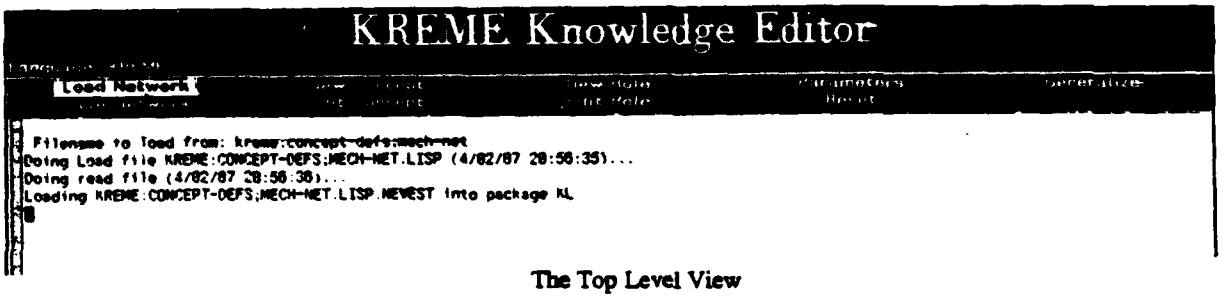
Figure B-2 shows the **Main Concept Editing View** in more detail, broken down into its component windows. As you will see, many of these windows appear in other views, as well. The Main Concept Editing View is the one you will probably use most of the time when you are editing frames. It contains the following windows (numbers in parentheses correspond to the labels in the figure):

The global command window (1) contains commands that may be invoked at any time while running KREME. These are described in detail in section B.1.3.

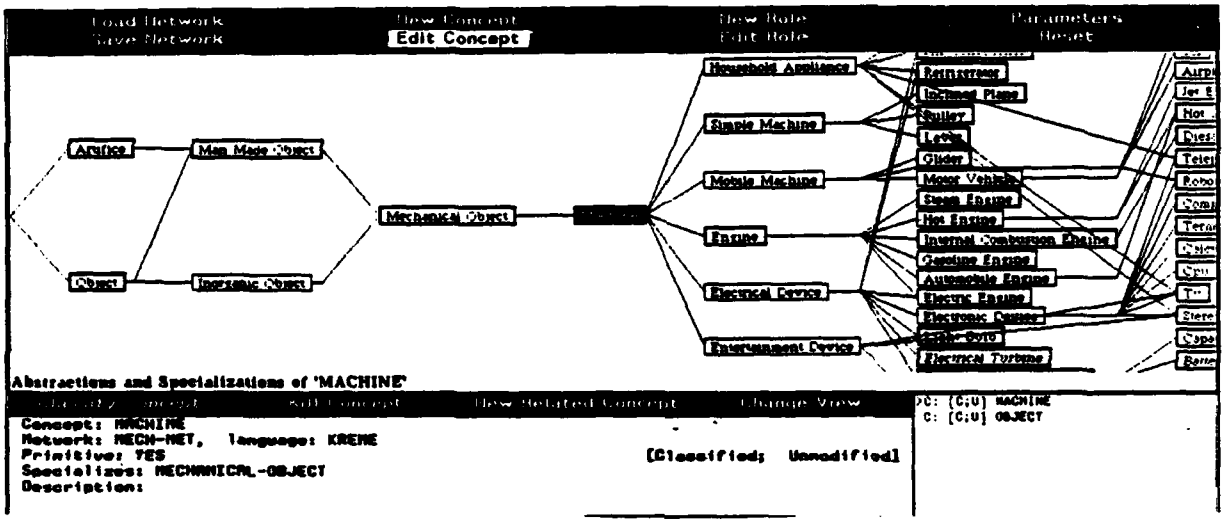
The editor stack window (2) shows the names of the things being edited and some information about their current edit state (e.g., whether they have been modified). The top item in the editor stack is the *current editor object*, the object which is the focus of the current view.

The state window (3) always displays pertinent information about the top item on the editor stack, which is called the *current editor object*. For concepts, the state window contains the concept's name(s), a line specifying whether the concept is *primitive* and whether the concept has been *classified* (defined) or not, and whether it has been *modified* in the editor since it was last classified. It also includes lines giving the concept's parents, and a textual description. Variants of this window appear whenever you are editing a concept, a role, or a rule packet.

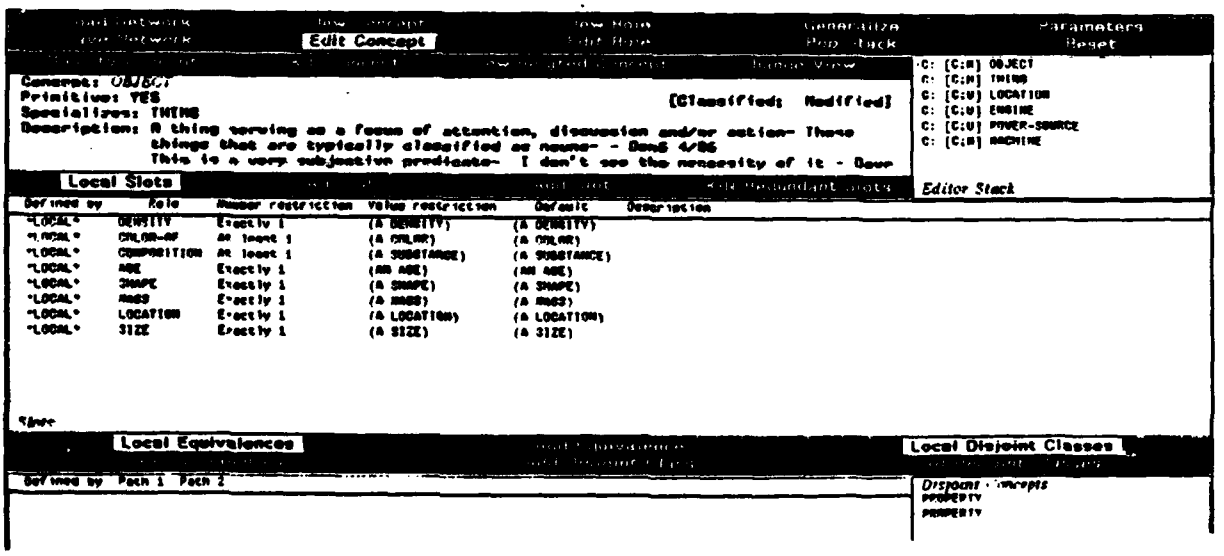
Figure B-1: KREME's Screen Editing Views



The Top Level View

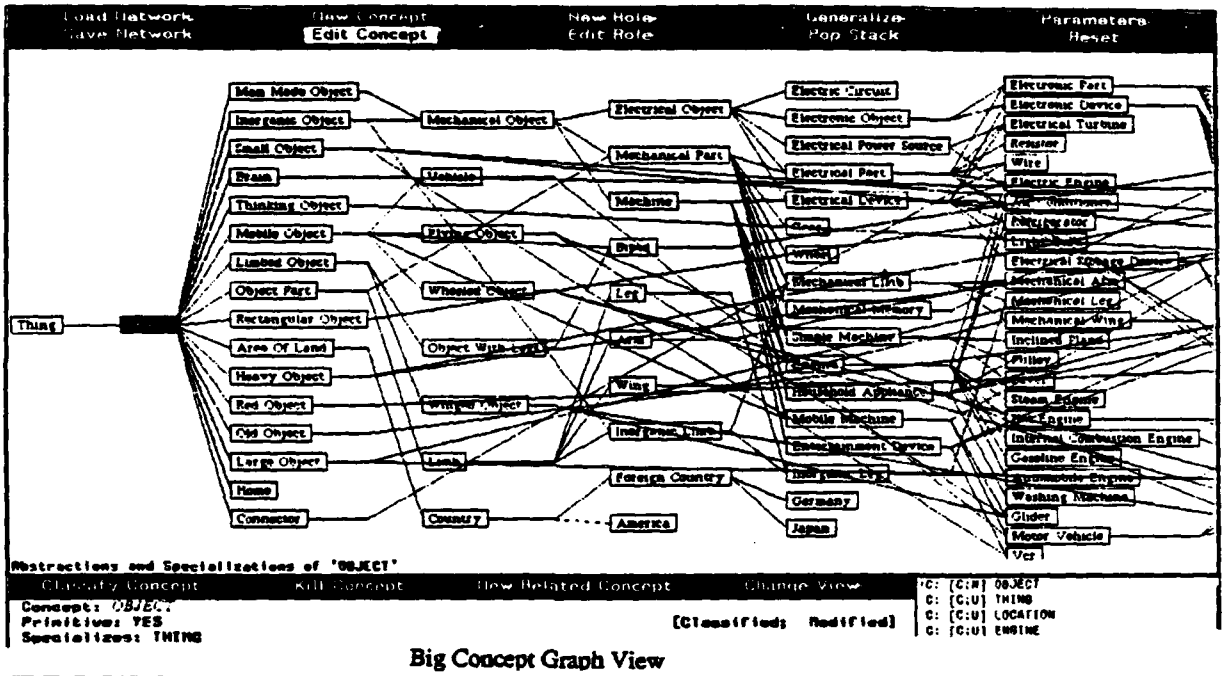


Main Concept Editing View

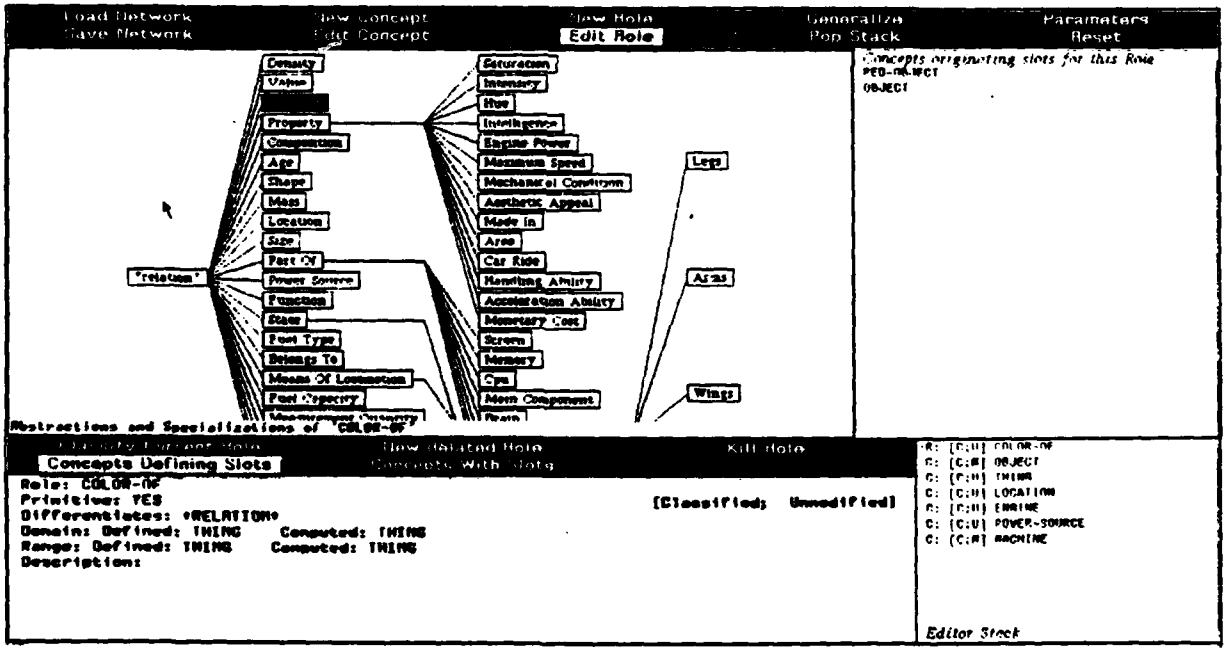


Alternate Concept Editing View

Figure B-1, continued.

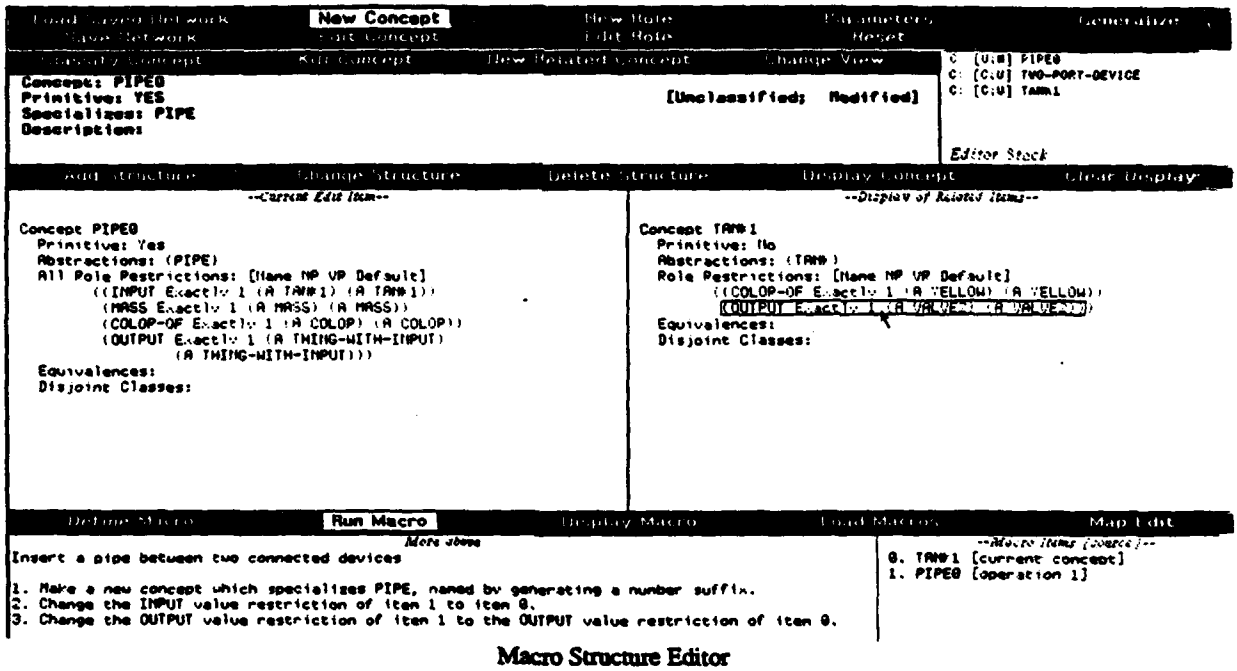


Big Concept Graph View

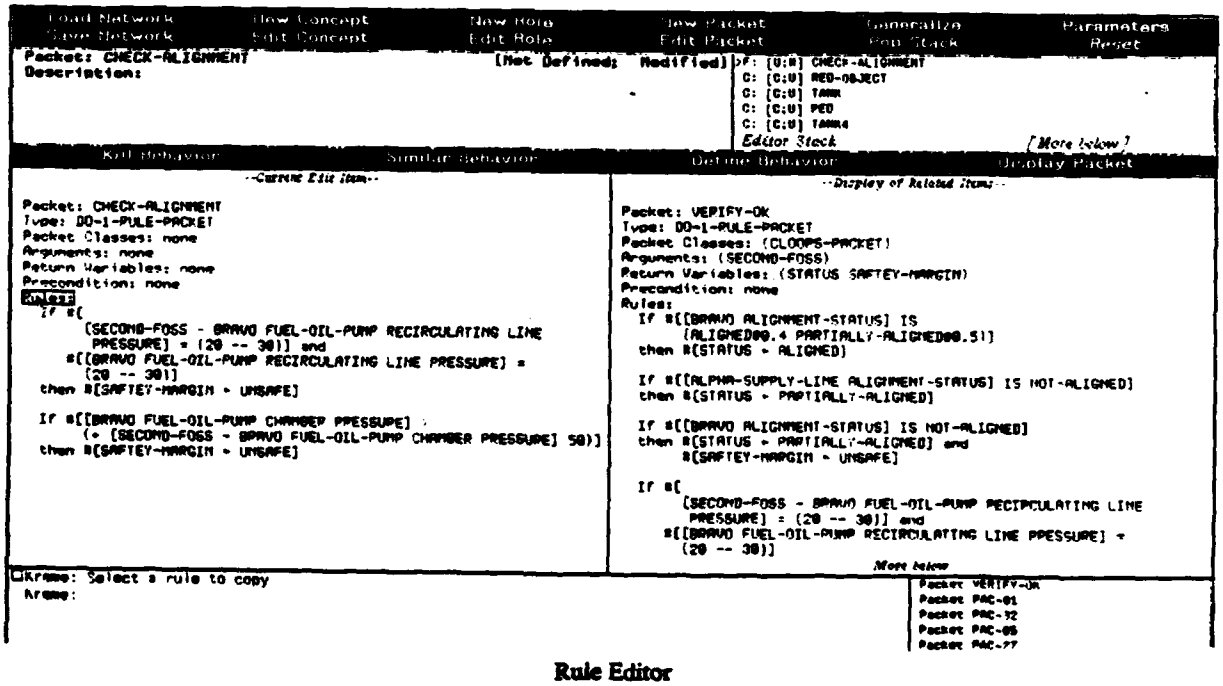


Role Editing View

Figure B-1, continued.



Macro Structure Editor



Rule Editor

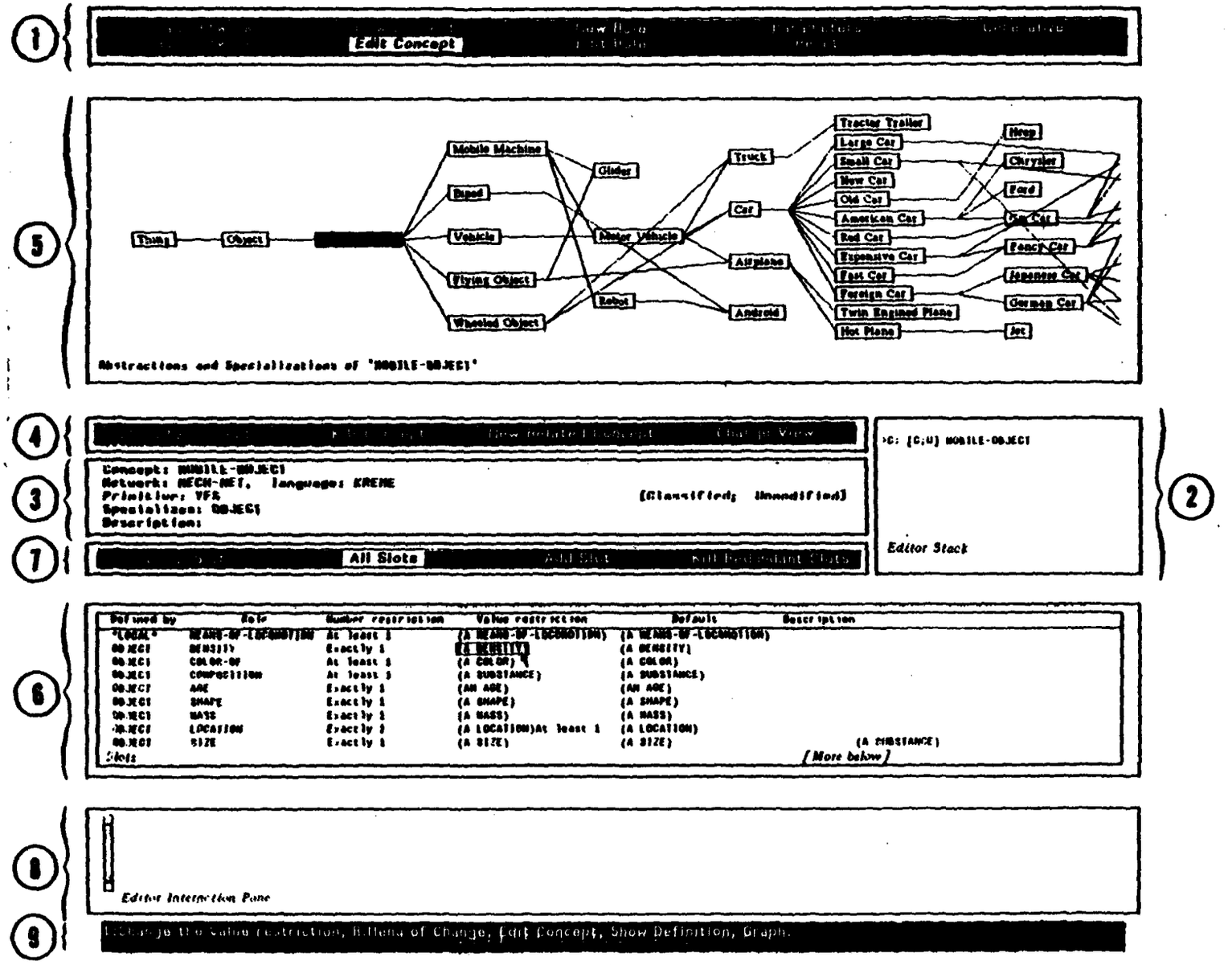


Figure 5-2: Windows in the Main Concept Editing View

The local command window (4) is a menu of commands specific to editing the kind of object displayed in the state window (the current editor object). It always appears directly above state window. When editing a concept, the commands are Classify Concept, Kill Concept, New Related Concept, and Change View, the last of which allows you to change to one of the other views available for editing concepts. When editing objects other than concepts, a somewhat different command window appears above the state window, containing commands useful for the type of object displayed.

The graph window (5) displays a dynamically updated graph of all of the abstractions and specializations of the current editor object. This view provides a constant visual display of the relative position of the object being edited in a hierarchy. Graph windows often appear when you are editing concepts or roles, or, in general, objects that live in hierarchies. The commands that are available within the grapher are described in detail in section B.2.2.2.

The table edit window (6) displays one of a number of tables describing a set of features that are part of the definition of the current edit object. The one displayed in (7) is the slot edit window, which has one line for each slot of the current concept. This is the normal table to see when editing a concept, although there are several others, which are described in section B.2.2.4. Columns in the slots table show the source (where it was inherited from) of the slot, the name of the slot (which is also the name of the *role* or relation that the slot represents), the slot's value and number restrictions, default value, and a textual description of the slot. General operations on table edit windows are described in section B.2.2.4.

The table edit command window (7) is a menu containing commands for changing and adding to the list of things displayed in the table edit window. The contents of this menu changes whenever the set of things displayed in the table edit window changes. When editing slots, commands are available to display the locally defined slots, display the full set of inherited slots, add a new slot, and kill all redundant slots (slots which are the same as inherited ones).

The Editor Interaction Window (8) is a Lisp Listener with a KREME command interpreter running. Both normal LISP expressions and KREME commands (like the ones displayed in command menus) can be issued here. This window is also used when KREME needs to ask the user for information. This window can be scrolled backward and forward through a history of the current session using the scroll bar at the left.

The Mouse Documentation Window (9) is always visible on Lisp machine screens. This is where you look to see what the mouse will do if you click one of its buttons. (See section B.1.2.) **IT IS VERY USEFUL -- ALWAYS GLANCE AT IT WHEN YOU MOVE THE MOUSE.**

Five of these windows are common to all views (except the Top Level View); the global command window, described above, appears everywhere, including the Top Level View. The editor stack window, the state window and the local command window appear everywhere but in the Top Level View. The mouse documentation window is part of the standard Symbolics interface, and so is always present. The graph window is currently used for displaying the hierarchies of concepts and roles only, although, in the future, we expect it will be used for other things as well.

There are a few of other types of windows, which will be discussed where they are most relevant. Among these are the structure editing windows that are used in the Macro Structure Editor, and in the Rule Editor.

B.1.2 Using the Mouse

Pointing with the mouse, and then clicking one of the buttons on the top of the mouse is the way virtually all commands are given to KREME. This includes all editing operations for browsing, adding, and modifying definitions, all commands to change what appears on the screen, and commands for loading and saving knowledge bases.

Wherever the mouse appears on the screen, something will happen if a button is clicked. You can always tell *what* will happen by looking at the bottom of the screen. There you will find a short one-line window called the mouse documentation window (See (9) in figure B-2 above.) that says what each mouse button will do. L: is what the left button will do. L2: tells you what will happen if the left button is clicked twice in succession. M:, M2:, R: and R2: describe the operations that will be performed by the middle and right buttons, respectively. Normally, you want to click the left button once to get the default behavior. The right button is always a menu of other operations that you may choose from instead.

In general, all visible references to an object can be pointed at, in order to view the object in more detail. For example, a concept or its name can appear as:

1. a node in a graph,

2. a value restriction or default in the slot description table,

3. the name of a parent in the state window,

4. as an item on the editor stack.

Whenever the mouse is over a [Command][9], or the name of some object, a rectangle is drawn around the name, and the content of the mouse documentation window changes. Commands are always executed using the left mouse button. Clicking the middle button displays some on-line documentation for that command. The right button is used to set optional command parameters before executing a command.

Whatever the form of the display, the displayed item will respond to the same set of operations when someone points at it, and those operations will be specified in the mouse documentation window. When the system requires the entry of a concept name, as when the [Edit Concept] command is clicked, the user may either type the name or

[9]Command menus always appear as black windows with white letters. Commands in this manual appear with boxes around them, as above.

point at any visible concept name.[10]

B.1.3 Command Menus

All commands that cannot be performed by pointing directly at an object appear in command menus which are associated with particular windows in each editor view. For example, the global command menu, which appears at the top of the screen in every KREME view, contains the following commands:[11]

- The Load Network command is used to read a previously developed knowledge base into KREME. After clicking on this command, you will be switched to the Top Level View, and prompted for a filename.
- The Save Network command is used to save the knowledge base, or a portion of the knowledge base that you have been editing. It prompts you for a filename which defaults to the last file read. (The default is used if you hit <Return>.) Clicking right on this command allows you to save branches of networks.
- Creating new concepts, roles, and packets of rules: the New Concept, New Role and New Packet commands, switch you to the default view for editing an object of the type specified, after asking you to name the object, and specify a little information about the object you wish to create on a pop-up menu form. The New Concept and New Role commands are described in section B.2 and an example of their use appears in Appendix C.
- Editing individual concepts, roles, and rule packets: the Edit Concept, Edit Role and Edit Packet commands all ask for the name of an existing object of the specified type, and then switch to a view in which that object can be presented for editing.
- The Generalize command is used to find generalizations. (See section B.5.)
- The Pop Stack command removes the top item from the editor stack[12]. (See section B.1.4.)
- The Reset command can sometimes be used to reset the editor, when it is stuck for some reason. Use the right button to erase the currently loaded knowledge base.
- The Parameters command is used to set some top-level parameters of the KREME environment. For normal operations, this command should ONLY be used to set the language syntax used to read concept definitions from files. At present, the only choices for this are KREME and NIKL. NIKL mode allows KREME to read definitions from files in NIKL syntax.

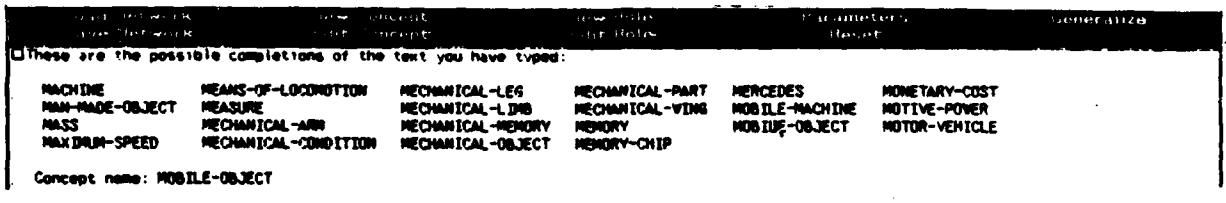
---

[10] A hyphenated name such as "MOBILE-OBJECT", will sometimes appear on the screen with a space between the two words ("MOBILE OBJECT"). However, when typing the name of something which appears as multiple words, put hyphens between the individual words.

[11] Occasionally, in some versions of KREME, there will be more commands than will fit in the space allotted for a command window. If there is a command that you know should be there, but can't find it, try to bump-scroll the command window and see if it is just hidden off of the screen. (If you don't understand this, just ignore it.)

[12] Analogous to the Kill Current Buffer command in EMACS.

When answering a question like the "Concept Name:" question that appears when you click the **Edit Concept** command, you may either type in a name or point at any concept name that appears on the screen. If you are unsure of the spelling, you may type part of the name and hit the <COMPLETE> key, which will cause KREME to try to fill out the rest of the name. You can also hit <COMPLETE> after typing just the first letters of hyphenated names. For example M-O<COMPLETE> will expand to MOBILE-OBJECT in our example network. Also, typing the beginning of a name and then c-? will cause all of the remaining possibilities at that point to be printed. You can then simply point at the one you want, as shown below.



B.1.3.1 Local Command Menus

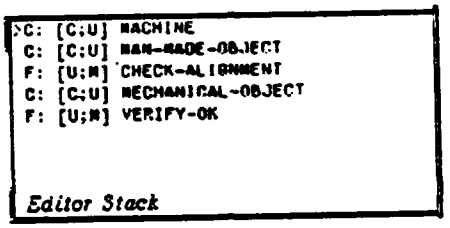
Anytime KREME is displaying a view of a particular kind of knowledge, the State Window displays the most basic facts about the object being edited. A command menu appears directly above the state window with some basic commands for the type of object displayed. For example, when editing concepts, the following menu appears:

In these local command menus, one will always find the command that makes permanent the definition that is currently being edited (a **Classify** command for concepts and roles), a **Kill** command (if applicable) to undefine the object, a command to make **New Related** objects like the current one, by copying the current definition permitting you to edit it, plus some miscellaneous other commands.

B.1.4 Buffers and the Editor Stack

KREME maintains a stack or list of the objects that have been edited, and constantly displays this list, indicating which objects have been modified and not reclassified. KREME behaves much like the text editor EMACS in this respect, since it maintains a distinction between things that have been *edited* (buffers in EMACS) and things that are *defined* (files for EMACS). For KREME, *defined* means classified.

This list of objects that have been edited in the current KREME session is displayed in the Editor Stack Window, an example of which is shown below:



Each line of the Editor Stack Window starts with a character indicating the kind of edit object it is (C: for concept, R: for role, F: for a rule packet, which becomes a *function* when run, an indication of the current edit state of the object, and the object's name. The top item in the stack (the line beginning with >) is the definition currently being viewed and edited. The user is free to modify this definition in any way without directly affecting the knowledge base. The edited definition is only made permanent when a command like Classify Concept is issued. When defining a new concept that has not yet been classified, the second line of the state window will show the words

[Unclassified; Modified]

and the corresponding (top) line of the Editor Stack will contain the symbols [U;M]. This refers to the fact that there is no permanent version of this definition yet (i.e., it is unclassified). Immediately after a definition has been classified, the State Window will display

[Classified; Unmodified]

At this point, the top line of the Editor Stack will contain the symbols [C;U]. If the object is then edited, the word Unmodified will change back to Modified.

The editor stack is always visible, providing a convenient method for browsing through a knowledge base. To make any definition item currently in the stack the top item, point at it and click the left mouse button. The object will be displayed in the same editor view as when it was last edited. Pointing at an item on the stack and clicking the right button pops up a menu that allows you to:

- Move to Top - make the object the current editor object, displaying it in the view in which it was last edited.
- Show Definition - display the (LISP form of the) object's definition.
- Graph - display a pop-up graph of the object's abstractions and specializations in a temporary window like the normal grapher window.
- Classify the object.
- Remove the object from the editor stack, without classifying it. The top item on the editor stack can also be removed using the Pop Stack command in the global command menu.

The Editor Stack Window, like most windows in KREME views, can be scrolled if more objects have been edited than will fit on lines in this window. When there are more items than will fit, the words *[More Above]* or

[More Below] will appear in the line with the words *Editor Stack*. To scroll, move the mouse into the window, and move it across the left edge slowly, until a double headed arrow appears, and follow the directions in the mouse documentation window. Alternatively, you can scroll a line at a time by moving to the bottom (or top) near the right side of the window and moving the mouse slowly downward (upward).

B.2 Editing Frame Knowledge Bases

This section deals with KREME Frames, and the KREME Frame Editor. The KREME Frame language is a close relative of the NIKL language that is the definitional (taxonomic) component part of KL-TWO and a descendant of the KL-ONE frame language [9, 15, 21]. This language provides an effective way of defining conceptual classes which live in a taxonomic hierarchy. The frames or *concepts* you define using the KREME Frame Editor serve as the *terminological component* of the knowledge based system being developed. That is, concepts are the *terms* to be referred to and manipulated by an *inference process*, perhaps defined by a set of *inference rules* developed using the KREME Rule Editor.

B.2.1 Definition of KREME Frames

In KREME, a frame is called a *concept*. Collections of concepts are organized into a rooted *inheritance* or *specialization hierarchy* sometimes referred to as a *taxonomy* of concepts. A single distinguished concept, usually called THING, serves as the root or *most general concept* of the hierarchy. Figure B-3 shows a simple specialization hierarchy. A concept (e.g., ELEPHANT in figure B-3) in one of these hierarchies *specializes* another concept (e.g., OBJECT) when the class represented by the concept is *subsumed by*[13] (is a subset of) the class represented by the other concept. Graphically, this means that the latter (OBJECT) appears an ancestor of the first (ELEPHANT).

A concept has a *name*, a textual *description*, a *primitiveness* flag, a list of *defined parents* (concepts that it is defined to *specialize*), a list of *slots*[14], a list of *slot equivalences*, and a list\of concepts that it is *disjoint from*[15]. In KREME, a concept may be subsumed by more than just the concepts that are its defined parents. Thus, classified concepts in a KREME hierarchy also contain distinct lists of those concepts that directly subsume it, and those which it directly subsumes or are its direct children.

The lists of slots, equivalences and disjoint concepts are collectively referred to as the *features* of a concept. If each concept can be thought of as defining a unique category, then features of the concept define the necessary

[13] For this reason, these hierarchies are sometimes called *subsumption lattices*.

[14] In NIKL slots were called *role restrictions*.

[15] One concept is *disjoint from* another if being in one class precludes being in the other.

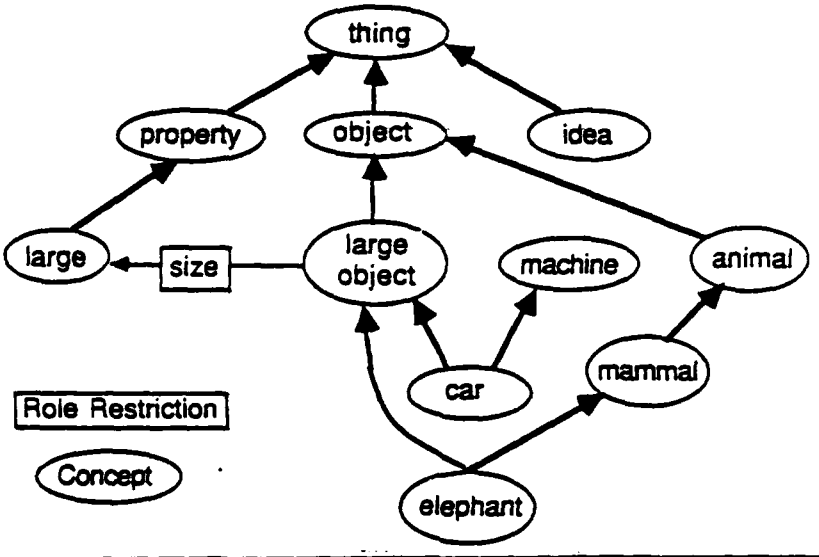


Figure B-3: A Simple Concept Taxonomy

```
(defconcept HOUSE
  :primitive t
  :specializes (building)
  :slots
     ((residents (a person) nil (a person))
      (front-door (a door) (1 1) (a door)))
  :equivalences
     ((main-entrance) (front-door))
  :disjoint (office-building apartment-building))
```

Figure B-4: LISP form of a KREME frame definition

conditions for inclusion in that category. If a concept is not marked as *primitive*, the features also constitute the complete set of sufficient conditions for inclusion in that category. A concept inherits all features from those concepts above it in the lattice (those concepts that subsume it, and thus are more general) and may define additional features that serve to distinguish it from its parent or parents. Figure B-4 shows the LISP defining form for a concept. Words prefixed by colons denote the type of feature. The :slots are specified as a list of lists of the form *<role-name value-restriction number-restriction default>*. Names of concepts in value restrictions and defaults are prefixed by *a* or *an*. Number restrictions are a list of *<minimum maximum>*, specifying how many objects of the type specified by the value restriction may be related to an instance of this concept. NIL here means "any number".

All of the slots defined for a concept, when taken together, form a description of the the attribute-value pairs that an *instance*[16] must have for it to be considered a member of the class defined by that concept. A slot consists of

[16] A token denoting a specific object in the world.

a role name, a value restriction, a number restriction and an (optional) default form[17].

The role name refers to an object called a *role*. Roles in KREME, are actually distinct, first class objects. Roles describe *relations* between concepts. A *value restriction* on a slot named by a role is a further specification or restriction of the *range* of the role, delineating the set of things that the concept that contains that slot can be related to. It is normally the name of some other concept. The *domain* of the role is a general characterization of the set of things that may use this role to relate objects to other objects. Put another way, it is the most general class of things that can use this role as the name of a slot. As first-class objects, roles form their own distinct taxonomy, rooted at the most general possible role, usually called *RELATION*. Figure B-5 shows a portion of a simple role taxonomy.

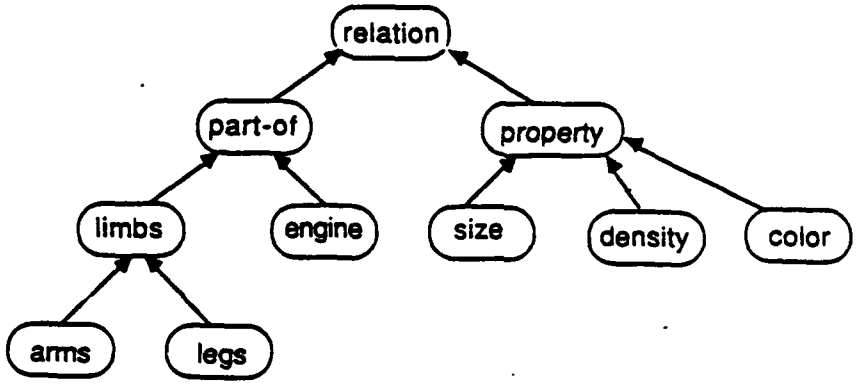


Figure B-5: A Simple Role Taxonomy

A role definition has a *name*, a *description*, a list of roles that it *specializes*, a *domain* and a *range*. In a formal sense, a role is a two-place relation that maps instances of concepts in its domain onto sets of instances in its range. The *domain* of a role is the most general concept at which the role makes sense. That is, it specifies the class of things for which the role can name a slot. The range of a role specifies the general class of concepts that can serve as values in slots defined using that role. All concepts filling slots whose name is a given role must be elements of the range of that role.

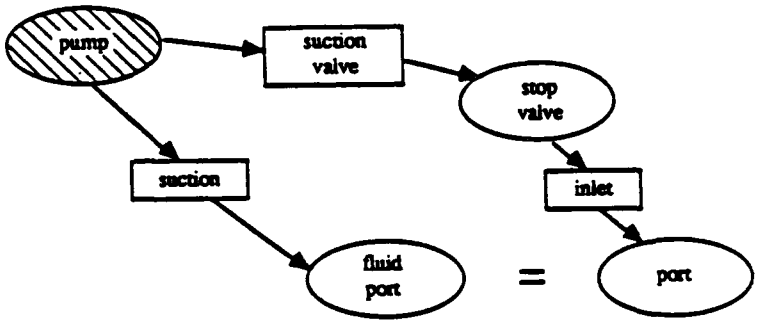
Each slot at a concept has as part of its definition a *value restriction*, which is the class of allowed values for that slot. The value restriction must always be a sub-class of the range of that role, and a sub-class of the value restrictions defined for that role at all concepts subsuming the one restricted. Value restrictions must be defined concepts.

Slots also include a *number restriction* that specifies the minimum and maximum (if any) number of things

[17]Defaults were not part of the definition of NIKL.

that may be related by the role to instances of the concept. For example, if all elephants have four legs, then the concept ELEPHANT might be defined to restrict the role LEGS to *Exactly 4* ELEPHANT-LEGs[18] A number restriction must be at least as specific as all of the number restrictions for the same role at any of the concept's parents. A number restriction of *Exactly 1* (min = max = 1) is more specific then a number restriction of *At most 2* (e.g., min = 0, max = 2).[19]

Equivalences describe slots that *by definition* refer to the same entities. They are defined as pairs of *paths* whose referents are the same concept. A *path* is a list of role names, the head (first) of which is the name of a slot at the concept defining the equivalence. Each subsequent slot name in a path must be a valid slot in the concept that is the value restriction of the previous slot in the path. The referent of a path is the value of the last slot in the chain. Figure B-6 shows a simple example of an equivalence.



The SUCTION of the PUMP is equivalent to the INLET of the SUCTION VALVE of the PUMP.

Figure B-6: A Slot Equivalence

Concepts marked as *primitive* (sometimes referred to as *Natural Kinds*) have no complete set of sufficient conditions. For example, an ELEPHANT must, by necessity, be a MAMMAL, but without an exhaustive list of the attributes that distinguish it from other mammals, it must be represented as a primitive concept. WHITE ELEPHANT, on the other hand, might be completely described by stating that it is a specialization of ELEPHANT, where the role COLOR was restricted to WHITE.

KREME Frames permit slots to have default values as well as value restrictions. If present, the default must be the description of some concept which satisfies the restrictions on the role at that concept. The default is used as a slot filler for instances of a concept that do not specify a value for the slot at instantiation time. Defaults are inherited from the most specific parent at which they are defined.

[18] E.g., Number restriction: min = 4, max = 4; Value Restriction: (an ELEPHANT-LEG).

[19] MIN = NIL is the same as MIN = 0. MAX = NIL is the same as MAX = *infinity*. A number restriction specified as a single number *n* has MIN = MAX = *n*. No number restriction (NR = NIL) means MIN = 0, MAX = *infinity*.

B.2.2 Using the Frame Editor

The KREME frame editor has five views, as shown in figure B-1, the **Main Concept Editing View**, the **Alternate Concept Editing View**, the **Big Graph View**, and the **Macro Structure Editor View**. Roles, which are also part of the KREME Frame language, are edited with the **Role Editing View**.

In this section, we will cover the details of the editing operations available in the first three of these views. The **Role Editing View** is covered in Section B.2.4. The **Macro Structure Editor** is covered in section B.4.

B.2.2.1 Editing in the Main Concept Editing View

Normally, when one creates a new concept or edits a concept for the first time, KREME makes that concept the top concept on the *Editor Stack*, and switches to display the Main Concept Editing View. There, KREME displays the concept as it exists at that time.

Figure B-7 shows how the graph window immediately displays all of the abstractions and specializations of the concept being edited, the state window shows its name, whether it is primitive or not, its edit state (classified or not, modified or not), its parents, and a textual description. The table window simultaneously displays all of the concept's locally defined slots.

In the remainder of this section, we will cover all of the operations *available by pointing* at all of the different parts of this frame editing view, as well as describing in detail the workings of the Grapher and Table Editing Windows, which also appear in many other contexts in the KREME environment. We begin with the Grapher.

B.2.2.2 The Grapher

KREME is equipped with a powerful, general graphing facility that rapidly draws lattices of nodes and links. Its main use is to provide a dynamically updated display of a concept or role and its place in the specialization or inheritance hierarchy. When editing a concept in the **Main Concept Editing View** or the **Big Concept Graph View**, or when editing a role, KREME automatically displays all of that object's abstractions and specializations, with more abstract objects to the left, and more specialized objects to the right of the current editor object.

As shown above in figure B-7, the graph is initially centered on the current editor object, which appears as a black node with white letters. All other objects appear as nodes with a white background. Objects that are defined as *primitive* are indicated by thicker box edges. Nodes that have been **modified** (edited but not reclassified) appear as nodes with a grey background.

Panning the Graph

An important feature of the grapher is that it can display graphs that are much larger than the window through which they are viewed. If you want to see a part of the network that is off the screen, point with the mouse at some point on the graph window not containing a node, and hold the left button down. The mouse cursor will change

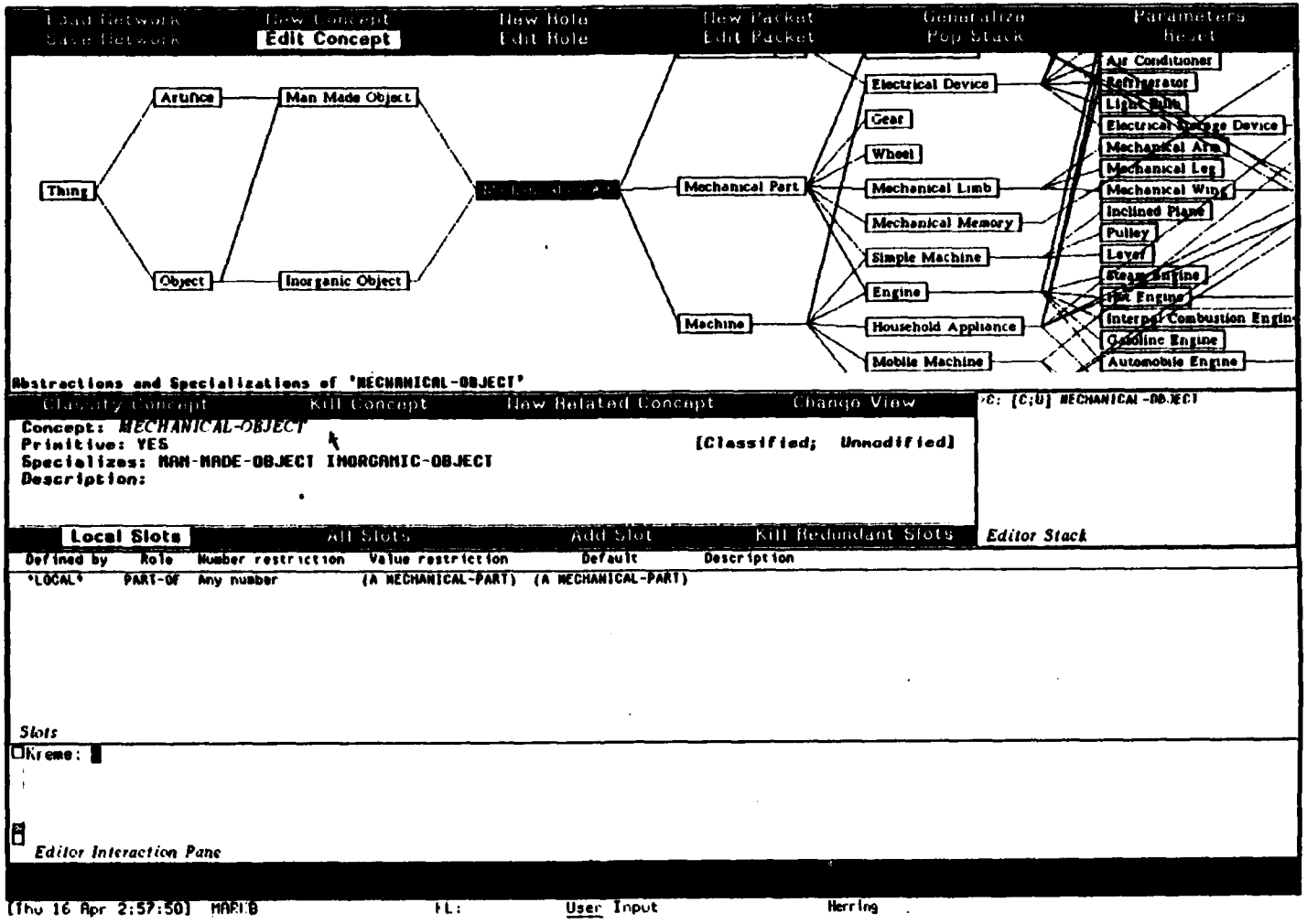
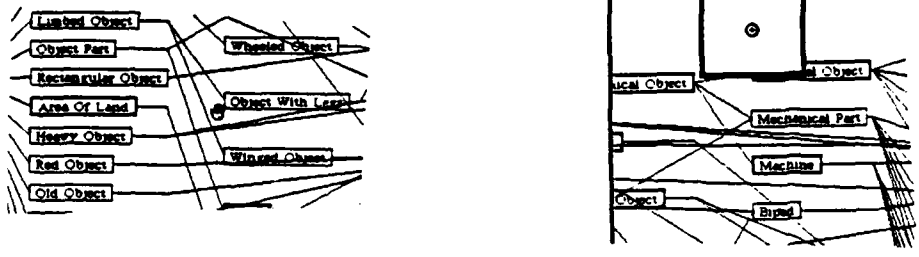


Figure B-7: The Main Concept Editing View

from an arrow to the shape of a hand. While still holding the left mouse button down, drag the mouse in the direction you wish the graph to move, and it will move smoothly as though you were pushing a piece of paper that was only partly visible in the space provided by the graph window.



Normal Panning

Speed Panning

Figure B-8: Panning the Graph

Another way to pan more quickly, called *speed panning*, is accomplished with the middle mouse button. Again, place the mouse over an unoccupied spot on the graph window, and push the middle button. A small square with a dot in it will appear. This is the "joy stick". While still holding the middle button down, move it a little bit off from the center of the box, and the graph will begin to move slowly in the direction you have moved. The further away from the center of the box you go, the faster the graph will move.

The Overview Graph

Now, click the right button once over an empty part of the graph window.

The **Graph Operations menu**, shown below, in figure B-9, will appear.

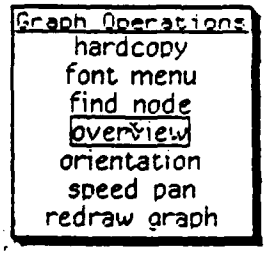
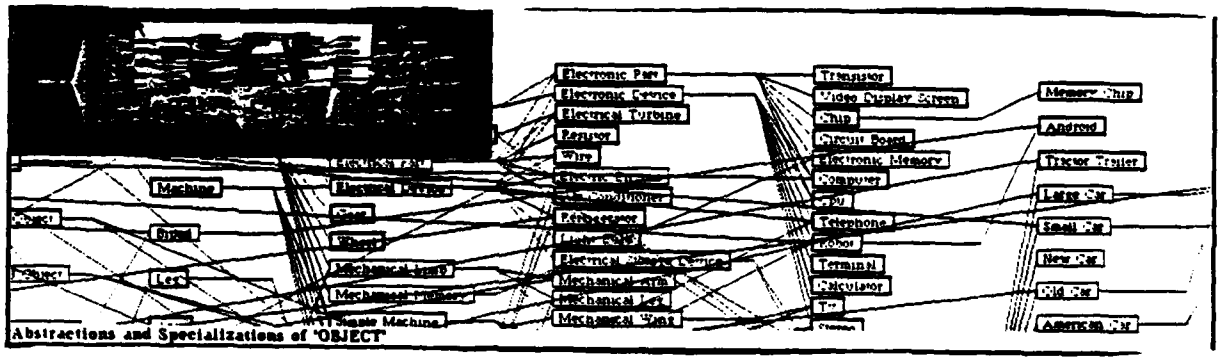


Figure B-9: The Graph Operations menu

We will discuss the other grapher options below, in section B.2.2.2. For now, click **overview**, and a miniature version of the full lattice will appear on in a black region in the upper left corner of the graph window.



This shows the full network displayed in the graph window, but with tiny nodes and links. The visible region of the graph will be indicated by a white rectangle inside the black one. Now pan with the mouse over the main graph window, and the white rectangle will follow your movements.

Now, move the mouse into the overview window. The nodes on the overview graph will be highlighted by a box as you pass over them, just as they are in the main graph window. All of the mouse operations available on nodes in the main window will also work on nodes in this window. (These operations will be covered in section B.2.2.2.) The name of the node is indicated in the mouse documentation window.

The overview window also can be used to pan the main graph window. Pointing (left button) to a spot on the overview that contains no node, causes the main graph to pan so that the upper left corner is at that spot. You can also hold and drag the mouse, and the graph will follow you.

To turn the overview off, simply bring up the Graph Operations menu, and click the command overview again.

The Graph Operations Menu

The other options in the Graph Operations menu show in figure B-9 are:

- **hardcopy** - Sends a copy of the full graph of the lattice to the printer.
- **style menu** - Allows you to chose a change to the font style and size of characters used for nodes on the graph. Smaller fonts are useful to see more of large networks at once.
- **find node** - Prompts for the name of an object on the graph, and centers that node on the graph window. It also draws a circle around the node so that you can find it more easily. The circle disappears as soon as the graph is panned.
- **overview** - Switches the overview graph between visible and invisible. The overview graph was discussed above in section B.2.2.2.
- **orientation** - Switches the orientation of the graph. Normally, the lattice is drawn from left to right. This command will cause the graph to be redrawn from the top of the screen down, and vice versa.
- **speed pan** - This command pops up the speed panning box without having to hold down the mouse button. In this mode, clicking any mouse button will make it go away.

- redraw graph - Redraws the current graph.

The Graph Node Command Menu

Normally, the KREME Grapher only displays the abstractions and specializations of the current editor object, rather than trying to display all of what is potentially a very large lattice. This was done intentionally, since KREME was designed to work with *very large knowledge bases*. Occasionally, however, one wishes to see more (or less) than KREME normally displays on such graphs. The Grapher provides a number of options to enable users to tailor what is displayed to their needs.

Whenever the mouse is over a node on a graph, the mouse documentation window shows the name of the node, followed by:

```
L:Edit this node.  M:Graph Relatives  R:Menu of Editing Options
```

Clicking the left mouse button causes KREME to make the object pointed to the top editor stack item. If you are looking at a concept graph, you will then be viewing the concept pointed to, a graph of its abstractions and specializations, and a table of its slots. This is an extremely convenient way of browsing through large concept networks quickly, and focusing on different portions of such a network. If, however, you wish to continue editing the concept you are currently viewing, but see more (or less) of the network around that concept or some other concept *on the same graph*, you can use the graph relatives menu found by clicking the middle mouse button over any graph node.

The graph relatives menu, exposed by clicking the middle button over a node, contains the following commands:

- Graph Parents - causes all abstractions of the node clicked on to be added to the displayed graph.
- Graph Children - causes all specializations of the node clicked on to be added to the displayed graph.
- Hide Children - causes all specializations of the node clicked on to be removed from the graph, unless they are also children of some other node.
- Hide Node and Children - causes the node clicked on and its children to be removed from the graph.

Editing a Network from a Graph

Clicking the right button over a graph node causes yet another menu of options to be exposed, the concept graph edit options menu.[20]

This menu contains the following options for concepts:

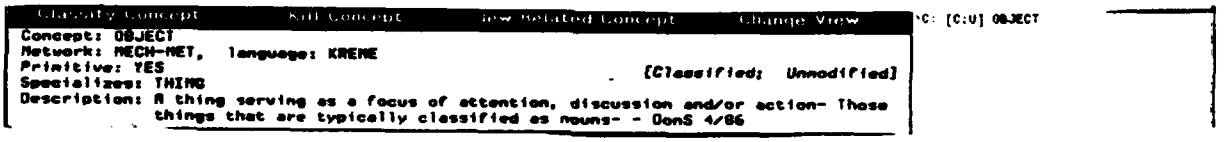
- Show Definition - This option causes the textual (LISP) form of the concept's definition to be displayed over the Graph Window.

[20] On graphs of roles, the role graph edit options menu appears, with essentially the same commands for roles, except as noted.

- **Kill Concept** - This causes the concept pointed to to be removed from the knowledge base. It has the same effect as the Kill Concept command in the local command menu window, except that it works when you are not currently editing the concept you wish to kill.[21]
- **Rename Concept** - This command prompts you for a new name for the concept pointed to, and immediately replaces all references to that name with the new name throughout the knowledge base. (See the Rename Concept command in section B.2.2.3.)[22]
- **Delete Parent** - This command prompts for the name of a parent (which you may give by pointing to the graph) and then deletes that parent from the list of defined parents of the concept initially pointed to. It also switches KREME to editing the concept modified, so that it can then be reclassified.
- **Add Parent** - This command also prompts for a parent, adds the concept named to the list of defined parents of the concept, and switches to editing the modified concept.
- **Splice Out Parent** - This command prompts for a parent, and removes that parent from the list of defined parents of the concept, replacing it with *that* concept's parents. Again, the editor is switched to a view of the modified concept.

B.2.2.3 Editing in the State Window

As described in section B.1.1.2, the state window of the Main Concept Editing View displays basic information about the concept currently being edited.



The top line displays the name of the concept, and any *synonyms* or alternate names for that concept. The name of the concept can be changed by clicking on the word **Concept:** and entering a new name.

The second line of the display shows whether the concept is defined as *primitive* or not, and whether the concept has been classified or modified since classification. Clicking on the word **Primitive:** causes the concept to be marked primitive if it was not, and vice versa.

The third line displays the both the *direct and defined parents* of the concept, after the word Specializes:. *Defined parents* are concepts that the user specifies as abstractions of the concept. *Direct parents* are concepts that may or may not have been *defined* as parents of the current one, but have been determined by the classifier to subsume the class denoted by this concept *and not have any specializations that also subsume this concept*. On the Concept Graph, the direct parents of a concept are the ones with direct links to it.

[21] There is no Kill command available on the role graph edit options menu.

[22] This command is called **Kill Role** on the role graph edit options menu.

This Specializes: list should be read as follows: Concepts that are unmarked are both defined parents and direct parents. Concepts that are defined parents but not direct parents are prefixed by a "-". Concepts that are direct parents but not defined parents are prefixed by a "+".

To add a parent to the set of defined parents of the concept, simply click the left button over the word Specializes: and type (or point to) the name of a concept that you wish to make a parent of this concept. To otherwise alter the set of defined parents , click the right button on the word Specializes:. You will be presented with a menu of the following options:

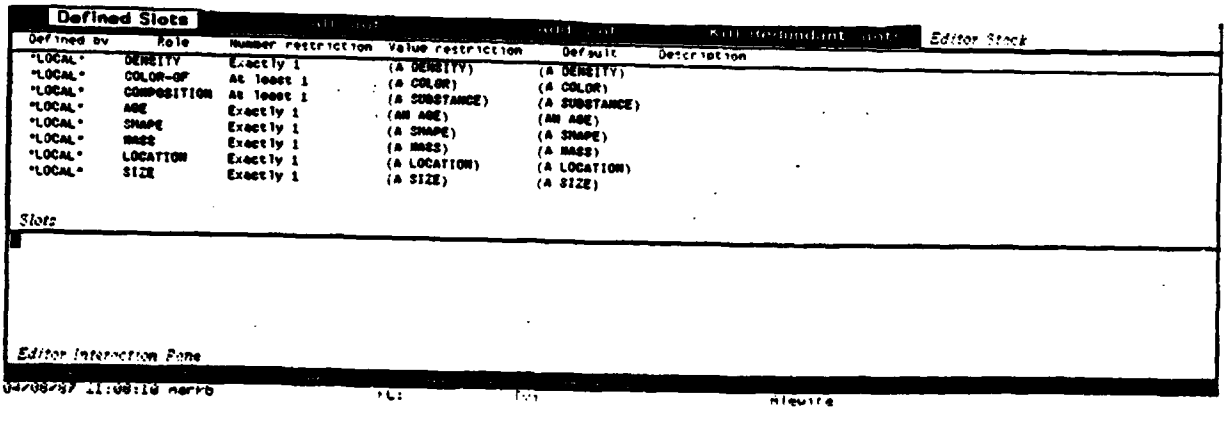
- Add Defined Parent - Prompts for the name of a concept to make a defined parent.
- Delete Parents which aren't direct - Allows you to point to defined parents that are not direct parents (i.e., those prefixed by a "-"), and have them removed from the list of this concept's defined parents.
- Make direct parents defined parents - This command causes *all* of the direct parents to become defined as parents of the concept.

The fourth and subsequent lines of the state window display the user-specified textual description of the concept, which provide a means of documentation. To enter a new description, click on the word Description: and you will be prompted for lines of text until you enter a blank line by just hitting <RETURN> or <END>.

B.2.2.4 Editing in the Table Edit Window

Normally, the table edit window in Main Concept View displays the set of Local Slots of the concept, that is, those slots which are defined locally by this concept and not inherited from above. The columns in the table are labeled "Defined by", "Role", "Number Restriction", "Value Restriction", "Default", and "Description".

Clicking (with the left mouse button) on the command All Slots in the table edit command window causes KREME to display both local and inherited slots. In this display, local slots are indicated by the word *LOCAL* in the "Defined by" column of the table. Slots inherited from a parent show the name of that parent. Slots formed by combining the value restrictions and/or number restrictions of several parents are indicated by the word *CLASSIFIER*. When the table window is displaying all of the concept's slots, you can return to viewing just the local ones by clicking the command Local Slots.



Whenever the Table Edit Window shows slots of the current concept, you can edit those slots or add new ones. To change the slot name, value restriction, number restriction, default, or description of a slot, simply click the left mouse button over the thing to be changed, and you will be prompted for a replacement. For all but number restrictions, the right button will pop up a menu that includes the commands [Change] the part of the slot pointed to, [Show Definition] of the concept or role pointed to, [Edit Definition] of that concept or role, or pop up a [Graph] of its abstractions and specializations. When pointing to the slot name, in the column labeled "Role", you can also [Rename Role], that is, change the name of the role, and all references to it in the knowledge base.

When the mouse is over a line in the slot table, and the entire line is encircled by a box, the right mouse button can be used to get a menu of [Delete Slot], [Copy Slot] to another concept, and [Move Slot] to another concept. For the last two, KREME prompts for the name for the concept to move or copy the slot to.

At any time, when you have started to make a change and are being prompted for a replacement value, you can hit the <ABORT> key to leave things as they were.

Adding New Slots

Whenever the slots table window is visible, as in the Main Concept Editing View, you can add new local slot definitions. A new slot is added to the defined slots of the concept with the [Add Slot] command. When this command is issued, the system prompts for a role name, a value restriction, a number restriction and a default form. Any of these items can be entered by typing or by pointing to the desired name or form if it is visible.

If a role or concept named in a role restriction or default does not exist the system will offer to make one with the name given, and proceed to pop up the defining form for that object. (See section B.2.2.5.) When you are finished filling out the form, click [x]Define, and KREME will continue to ask for the rest of the new slot's features.

When you have finished adding and modifying the slots of a concept, you should always make the changes permanent with the [Classify Concept] command. For an extended treatment of this command, see section B.3, below.

Modifying the Table Edit Window

The appearance of Table Edit Windows can be modified in several ways. The tables are scrollable in both the up-down and left-right directions. Simply "bump" the mouse against the top or left sides of the window until the double headed arrow appears and follow the directions in the mouse documentation window.

If you do not wish to see some columns of the table, they can be selectively removed by clicking the middle button in the line displaying the column headings (when the double arrow is *not* showing). You will be presented with a menu on which you can tick off the columns that you wish to see and not. When you are satisfied click the box marked [x]Do It. If you do not wish to go through with the change, click [x]Abort.

Changing the Contents of the Table Window

Since there is not enough room in the Main Concept Editing View to display all of a concept's defining features at one time, the contents of the Table Edit Window can be changed to display those other features. To do this, you must use the mouse to find the table window contents menu. This menu is available wherever there is nothing else under the mouse while still inside the table window. You will know you have found it because the mouse documentation window will show the words:

```
R: Change the contents of this table.
```

The best places to look are to the right of the "Description" column, and anywhere in the line of column headings (when the double headed arrow is *not* showing). Clicking the right button, you will see the following menu options:

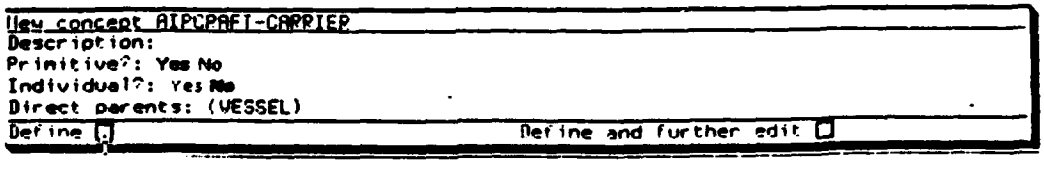
- Slots - Displays the table of this concept's slots, as described above.
- Inverse Restrictions - Displays a table, essentially like the slots table, but of all of the slots displayed are slots of other concepts that use the current concept as their *value restriction*. This table is useful when you are tracing references to a concept in other concepts. When this table is displayed, the table edit command window will be empty. Some of the editing options described for the slots table will not work here.
- Slot Equivalences - This table displays the slot equivalences of the current editor concept. This table has only three columns, "Defined by", "Path 1" and "Path 2". The two paths are designated as denoting the same object. Since slot equivalences can be inherited, their source is also indicated in the table, in the column "Defined by". When this table is visible, the table edit command window will show the commands Local Equivalences, All Equivalences, and Add Equivalence. The first two just change which equivalences are displayed. The last prompts for two slot paths that should be made equivalent.
- Disjoint Concepts - This table is just a one column list of all of the concepts that are defined to be disjoint from the one currently being edited. When this table is visible, the Table Edit Command Window will display the commands Add Disjoint Class, Local Disjoint Classes, and All Disjoint Classes.

B.2.2.5 Operations on Concepts

Making new concepts and roles

Clicking on the New Concept or New Role command in the global command menu is the simplest way to make a new concept or role.

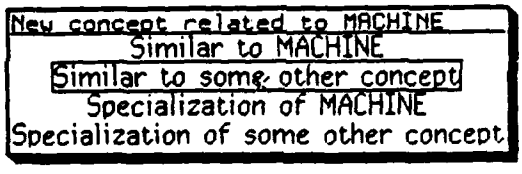
When making a new concept, KREME will prompt for the name of the new concept, and then a pop-up form will appear that looks as follows:



Here, you may specify a description of the new concept, whether it is primitive or not, whether it is an individual (a special kind of primitive class that has only one member, like the color RED), and a list of its direct defined parents, which must be a list enclosed in parentheses.

When you are done, click either the box labeled [x] Define or [x] Define and further edit if you wish to make the new concept the current editor object in order to do things like adding new slots before classifying it.

Another way to make a new concept, one that is similar to another concept, is to use the **New Related Concept** command in the **local command menu**. This command allows you to choose from a pop-up menu whether you want to make a new concept that is similar to the currently visible concept or to some other concept, a specialization of the current concept or some other concept, or a specialization of several concepts.



If you choose to make the concept *similar* to some existing concept, KREME makes up a concept definition that is identical to the concept you specify, but with the new name, and then allows you to edit it to make it somewhat different. If you choose to make the concept a *specialization* of some existing concept, KREME automatically makes the parent of the new concept be the one you are specializing.

The command **New Related Role** in the **local command window** of the **Role Editing View** works essentially the same way.

Killing Concepts

Changing the name of a concept or role directly affects the network. The name of the concept definition, as well as the name of the corresponding classified concept (if there is one), is changed. All pointers to the concept (as a parent of other concepts, in value restrictions, as the domain or range of roles etc.,) are automatically updated with the new name both in the classified network and in all editor buffers.

The **Kill Concept** command splices a concept out of the taxonomy. With this command, the children of a

concept will be connected to all of its parents. Any concept that used to define the concept as a parent is reclassified. If the concept was used as a value restriction, the editor tries to find an appropriate parent to substitute for the killed concept. Because this attempt is not always successful, user interaction is sometimes required.

Deleting redundant slots

To delete any defined slots whose definitions are the same as the inherited definitions, click on the Kill Redundant Slots command in the slots table command menu window, above the the slots table edit window. This operation alters the definition of the concept, but not its classification or completed description. (Classification will be discussed in detail in section B.3 below.)

B.2.3 Alternate Concept Views

Three other views are currently defined for concepts, and one view for roles. Two concept views display windows not normally visible in the Main Concept Editing View (see figure B-10), while the third is the Macro Structure Editor to be discussed in section B.4 below.

To change to an alternate concept view, use the Change View command in the local command menu window above the state window. Clicking on Slots and Equivalences brings up the first window in figure B-10. This view shows an enlarged slots table edit window, the disjoint concepts window, and the slot equivalences window together, along with the editor stack window, the state window, and associated command menu windows.

Clicking on Large Concept Graph shows the second view of figure B-10. This view uses most of the screen to display the specialization hierarchy, together with the State Window and Editor Stack Window.

B.2.4 Editing Roles

The Role Editing View (figure B-11) appears whenever the Edit Role or New Role commands are issued from the global command menu, when pointing at the name of a role appearing as the name of a slot, or pointing at a previously edited role appearing in the Editor Stack.

The Role Editing View contains a window showing a graph of the role specialization network, highlighting the currently visible role, another displaying a list of the concepts that restrict the role, and the role state window. The local command menu window for roles contains the commands:

- Classify Current Role - Classifies or makes permanent a new or changed role definition, inserting it into the role hierarchy, and checking that any concepts that use that role to name a slot satisfy the domain and range constraints specified.
- New Related Role - Creates a new role that is similar to or a specialization of the current (or some other) role. Its operation is analogous to the New Related Concept command.

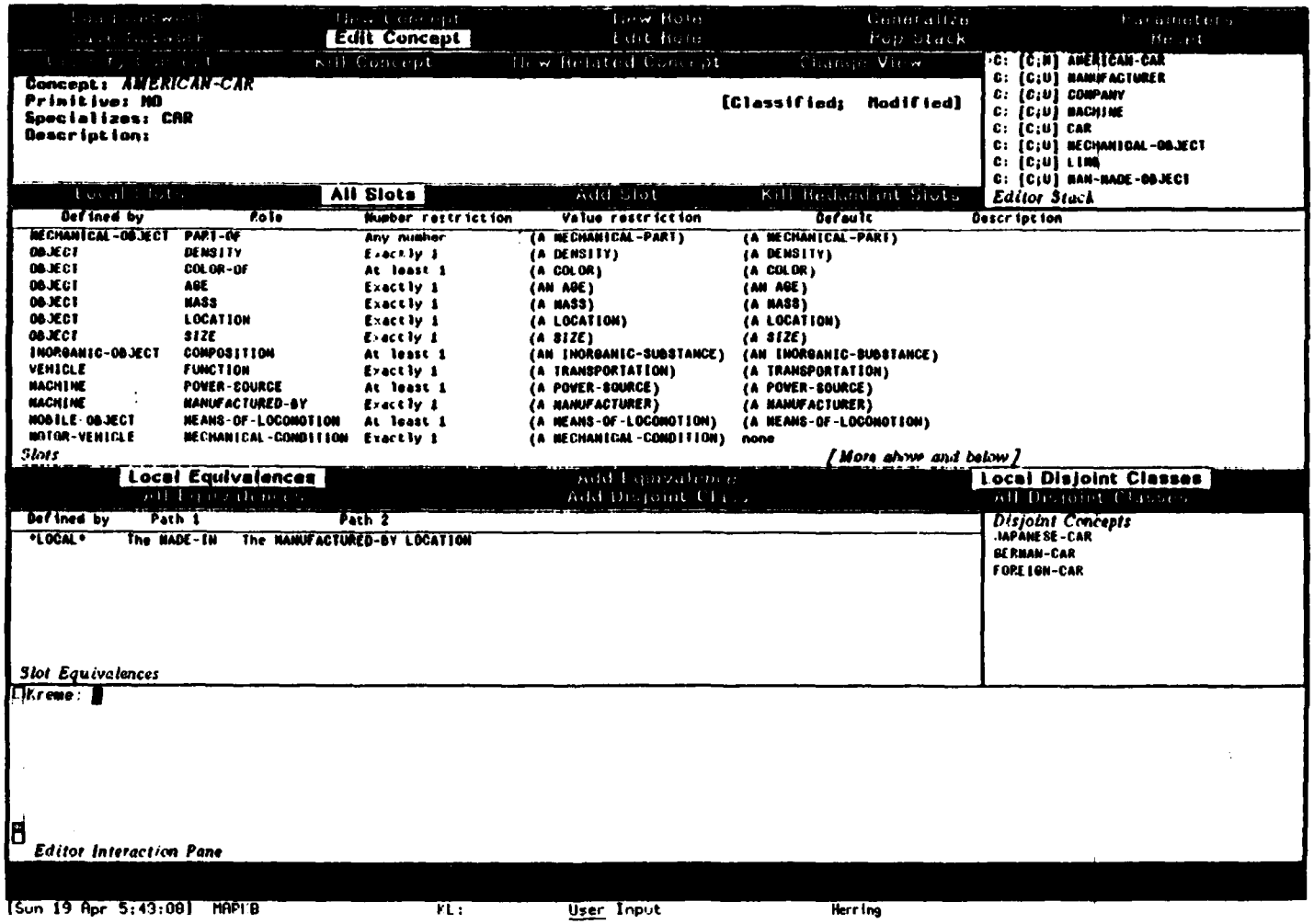


Figure B-10: Alternative Concept Editing Views

- Concepts Defining Slots - Changes the window showing the list of concepts using this role as a slot name to include only ones that *locally define* those slots, as opposed to inheriting them.
- Concepts With Slots - Changes the same window to display *all* concepts that have a slot with this role name.

B.2.4.1 Editing in the Role State Window

A similar set of operations exists for editing the basic features of roles in the role state window as exists in the concept state window.

- Role: invokes a command to change the role's name, and all references to it in slots.
- Primitive: toggles the primitiveness of the role.
- Differentiates: allows you to add a parent to the role. Clicking the right button over the word Differentiates: brings up the menu of Add Defined Parent, Delete Parents which aren't direct and Make direct parents defined parents, all of which work as described for concepts in section B.2.2.3.
- Domain: or Range: prompts for a replacement value for the defined domain or range of the role, respectively. Clicking the right button on one of these words gives you a menu of options including those discussed above and, when appropriate:
  - Defined domain equal computed value which makes the defined domain be the same as the *classified domain*, which is intersection (really the conjunction) of the role's defined domain and the domain of its parent role(s) (if any).
  - Defined range equal computed value which makes the defined range be the same as the *classified range*.

B.3 The KREME Classifier

B.3.1 Introducing the Classifier

One of the most time consuming tasks in building knowledge bases is maintaining internal consistency. Adding, deleting and modifying slots and parents in a frame taxonomy may affect the subsumption relations between frames and, perhaps more importantly, may alter the sets of properties inherited by more specific frames. The possible consequences of a change in one part of a network grows rapidly as taxonomies get larger. Consequently, the size and complexity of knowledge bases is limited by the extent to which automatic means are provided for consistency checking.

The KREME *classifier* helps the user maintain consistency between the definitions of all concepts defined in a KREME Frame knowledge base. It must be invoked (by the user) whenever a concept or role is defined or redefined. The classifier first gathers all of the features to be inherited by a concept, and then determines exactly where the concept should be placed in the specialization hierarchy.

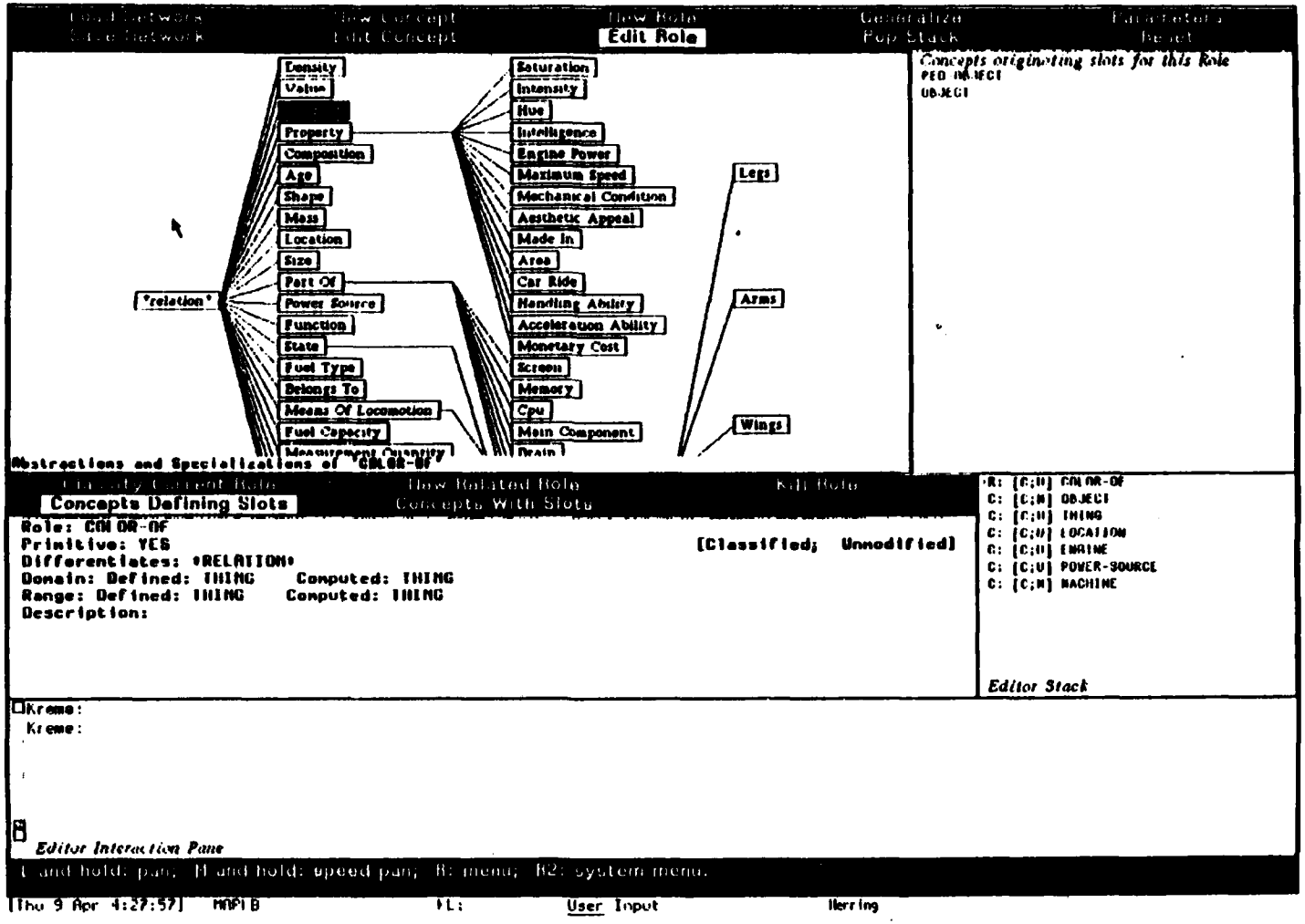


Figure B-11: The Role Editing View

The classifier should always be invoked once you are satisfied with a concept or role's definition. It is the mechanism by which KREME makes the new definition permanent, and inserts it into the knowledge base. To classify a definition, click on the Classify Concept command, in the local command menu window (the Classify Role command if you are editing a role) or use the Classify command in the pop-up menu available by clicking the right button on some object in the Editor Stack Window. KREME will determine the *completed definition* (a definition plus all its effective, inherited features) of the object and use that information to determine the object's relative location in the subsumption hierarchy of all previously classified definitions by deducing what the new concept's most specific parents and least specific children are. The system also checks to see if other concepts or roles need to be reclassified because of the new definition. If so, KREME will continue until it has reclassified every object that might have been affected.

After a concept has been classified or reclassified, KREME immediately displays the effects of classifying the definition. Visible abstraction-specialization graphs are redrawn, showing how the arrangement of parent-child relationships throughout the taxonomy has changed. On these graphs, links added or deleted by the classifier will seem to appear or disappear instantaneously. For example, the classifier makes sure that the *direct parents* of a concept includes only defined parents with no children that subsume the concept[23].

B.3.1.1 Completion

Completion refers to the basic inheritance mechanism used by the KREME classifier to install all inherited features of a concept in its internal description. Given a set of defined parents and a set of defined features, the completion algorithm determines the full, logically entailed set of features at a concept (or role). Completion always occurs before classification or reclassification of a role or concept.

A concept inherits all the value and number restrictions on every slot from all of its parents. For each uniquely named slot at each concept, a single number and value restriction is created that conjunctively combines all restrictions for that slot from the local definition of the slot and the definitions at every parent. The effective value restriction is either the single most specific of all the value restrictions for that role at the concept, or a conjunction of all of them, if no single one is subsumed by all the others. The effective number restriction for each slot is similarly determined by intersecting the number ranges in all of that slot's inherited number restrictions.

Complications arise when more than one parent concept defines the same slot, and no restriction on that slot is more specialized than all of the others. Figure B-12 illustrates one way this can occur; when the most specific value restriction is inherited from one parent (ANIMAL) and the most specific number restriction is inherited from another parent (4-LIMBED-THING) to form the restriction of LIMBS at 4-LIMBED-ANIMAL.

Figure B-13 shows another example of completion in which the resulting value restriction must logically be the conjunction of several concepts. Since ANIMAL-WITH-LEGS is an ANIMAL, and a THING-WITH-LEGS all

[23]Only the most specific descendants of defined parents that still subsume the concept are direct parents, and appear connected by direct links to it on the graph.

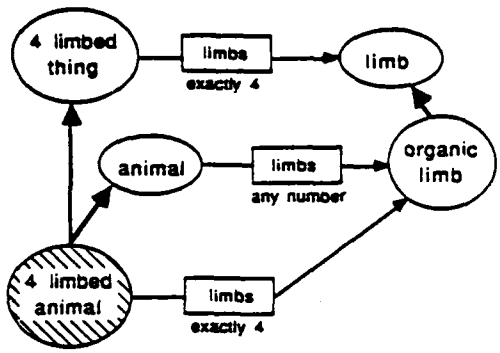


Figure B-12: Inheriting Number and Value Restrictions

of its LIMBS must be both ORGANIC-LIMBs and LEGs. If the concept ORGANIC-LEG, specializing both ORGANIC-LIMB and LEG, exists when ANIMAL-WITH-LEGS is classified for the first time, the classifier will find it and make it the value restriction of the slot LEGS at ANIMAL-WITH-LEGS. If it does not exist, the classifier stops and asks if the user would like to define it.

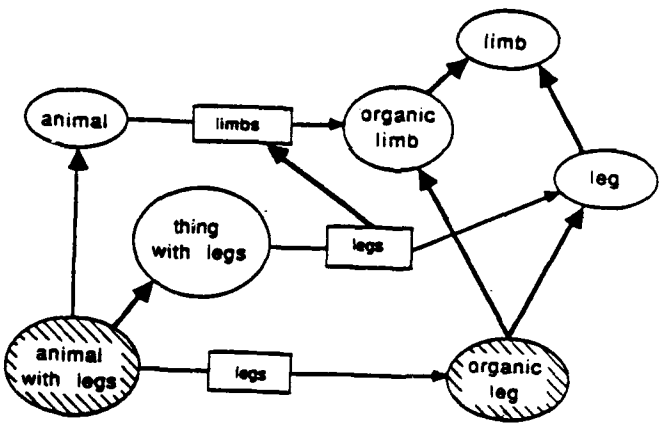


Figure B-13: Combining Value Restrictions

In general, whenever a value restriction can only be defined as a conjunction of several concepts, KREME offers to form a concept representing the conjunction, and asks for a name for the new concept. These new concepts, called *CMEETs*, must be named by the user.

B.3.1.2 Interactions with the Classifier

As indicated above, the KREME classifier sometimes needs to form new concepts in order to satisfy some logical relationship or determine the effective restriction on the range of a role. These classifier required conjunctions are called *CMEETs*.

CMEETs are formed when the classifier is trying to determine the effective value restriction for a slot, or the effective domain or range of a role. At such times, KREME enforces the restrictions that the concept or role inherits from above, while incorporating the locally defined constraint. KREME requires that the value restriction of any slot is at least as specific as all of the inherited value restrictions on that slot (and the range of the role naming the slot). Technically, the effective restriction on a slot is always the conjunction of (e.g., the class denoting the intersection of) all inherited restrictions and the locally defined restriction on the slot. Thus, if one defined the concept FROG as an ANIMAL-WITH-LEGS, and defined the slot LEGS to be restricted to (a FROG-LEG) without defining FROG-LEG as both a LIMB and ORGANIC-LEG, KREME would ask to make a CMEET that combined all of these classes. Since you probably would want to change the definition of FROG-LEG rather than create a new term, KREME allows you to say this, rather than create the new concept.

The third major case in which CMEETs are formed is when a value restriction is not subsumed by the defined range of the role that names the slot. Thus, if the *role* ENGINE-OF had range (AN ENGINE) and the *slot* ENGINE on CAR was defined with value restriction (A CAR-MOTOR), which had (perhaps accidentally) *not* been defined as a kind of ENGINE, KREME would ask if you wanted to define the CMEET (AND* (A CAR-MOTOR) (AN ENGINE)). Again, you probably want to must make CAR-MOTOR a kind of ENGINE.

Lastly, CMEETs are formed when determining the effective domains and ranges of roles that are children of other roles. However, it only happens if you define a role to specialize another role, and are not careful to make sure that the domain and range you specify are subsumed by the domain and range of the parent, respectively. In any case, KREME will let you know, and enable you to fix it, one way or another.

B.3.1.3 Options when asked to form CMEETs

While forming the appropriate conjunction is the logically correct thing to do to ensure consistency of the knowledge base *as then defined*, it often turns out (as suggested in the preceding section) that the conjunction suggested by the classifier is needed because one of the concepts to be conjoined has been improperly defined. In particular, a CMEET condition most frequently arises because the concept used as the value restriction of a role in the concept being classified is not subsumed by the restriction for the same role at a higher concept, and the restriction must logically satisfy both constraints. This is illustrated in figure B-14. The figure shows TWO-PORT-TANK defined as both a TANK and a TWO-PORT-DEVICE. Each of those concepts restricts the role INLET-VALVE. The classifier finds that the restriction for slot INLET-VALVE at TWO-PORT-TANK must be both a VALVE and a STOP-VALVE, given the restrictions of that slot at TWO-PORT-TANK's parents. Since STOP-VALVE was not defined as a kind of VALVE, the conjunction is not the single concept STOP-VALVE, and so the classifier asks if it should create a new concept, the CMEET of VALVE and STOP-VALVE.

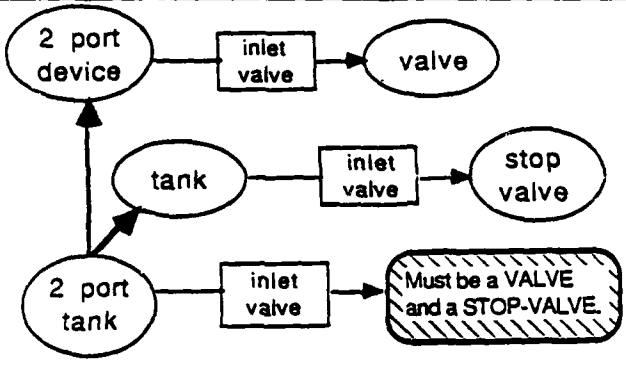


Figure B-14: Discovering a missing subsumer by a CMEET check.

Whenever the KREME classifier requires that a CMEET be formed, it stops and queries the user, explains the situation and requests a name for the concept to be formed for the conjunction, and enumerates several alternative options, as shown below.

You have several options at this point. If all of the concepts are defined correctly, and the proposed CMEET correctly describes the required restriction, simply enter a name for the new concept and classification will continues. If the problem really lies with an existing definition, as is the case with VALVE and STOP-VALVE, you can choose an alternative course of action. rather than introducing a useless new concept. Most often, the correct action is to alter the subsumption relations between the named concepts so that one of them is subsumed by the others. This is done simply by naming one of the concepts to be conjoined instead of giving a new name. In our example, the user would simply type STOP-VALVE, in response to the query. The classifier would then make STOP-VALVE a kind of VALVE and continue classifying TWO-PORT-TANK, resulting in the relations shown in figure B-16.

This interaction effectively allows a user to correct an oversight in a previously defined concept's definition at the point the error is detected by the classifier.

If forming the CMEET is the appropriate action, simply enter a name for the new concept; classification will continue. If you do not intend to form this new concept, name the more specific concept. This alters the subsumption relations between the concepts to be conjoined, so that one of them is subsumed by the others.

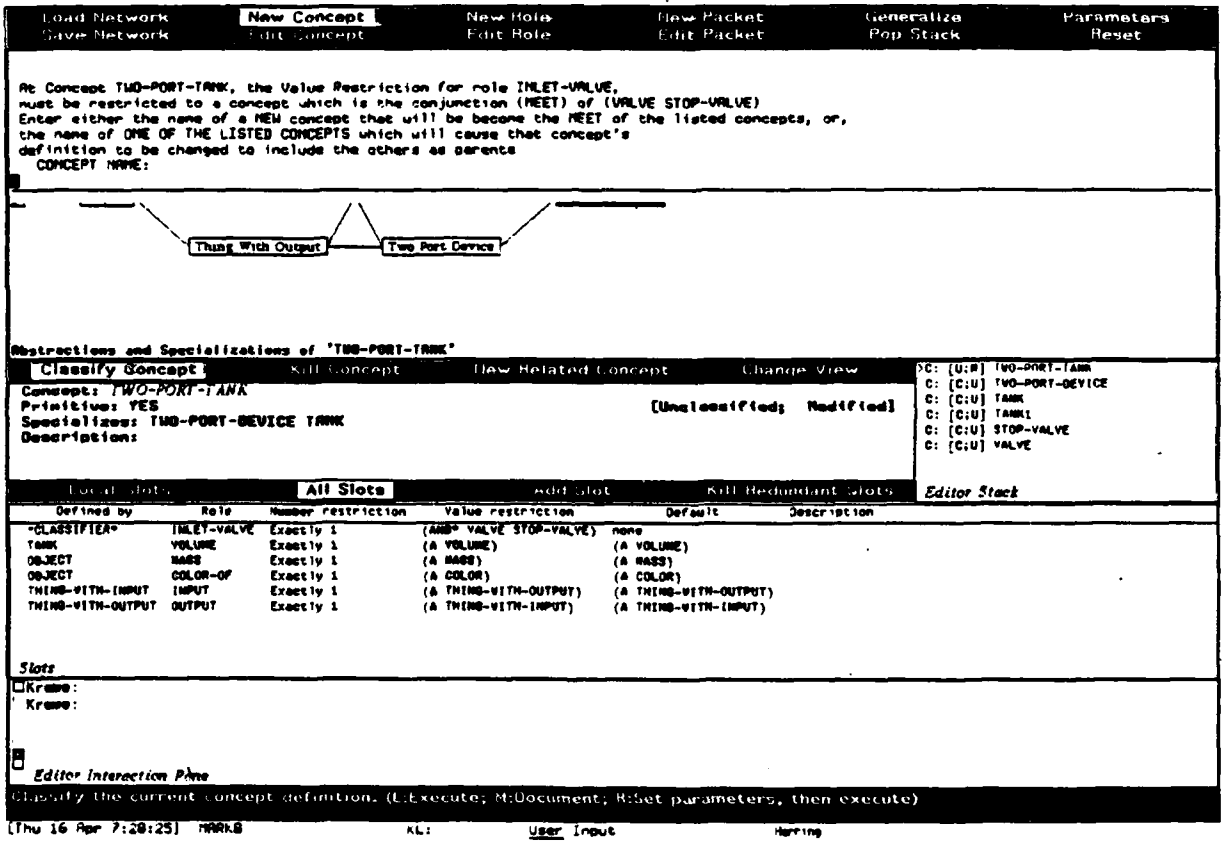


Figure B-15: Altering STOP-VALVE to correct a CMEET error.

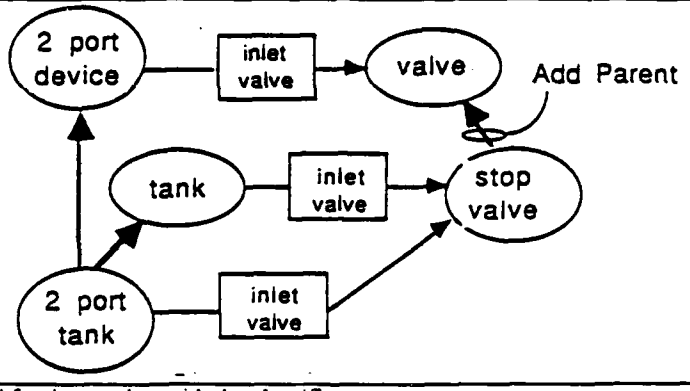


Figure B-16: After interaction with the classifier.

B.4 The Macro and Structure Editor

B.4.1 Macro Editing of Knowledge Bases: Background

Quite frequently choices about representations made early on in the development of a Knowledge Base proves to be inappropriate, and massive editing is required to convert the accumulated representation base. A macro facility makes these decisions easier to reverse, and therefore, less disruptive and costly.

In order to express and package conceptually clear *reformulations* of concepts and other representations, as well as develop new concepts from old ones, KREME provides a macro facility for reformulations. This facility can be expressed as sequences of standard, low-level editing operations which define editing macros to be applied to sets of concept definitions by giving a single example.

B.4.2 The Macro and Structure Editor View

One of the views available when editing concepts in KREME is the Macro Structure Editor View. This view (See figure B-17.) provides an alternative set of display and editing facilities for concept definitions. The view provides two windows for the display of stylized defining forms for concepts. The current structure edit window displays the definition of the current editor object concept (the top item on the editor stack). The display structure window is available for the display of any number of other concepts. Any concept which is visible in either window can be edited, and features can be copied from one concept to another by pointing. Both windows can be scrolled to view additional definitions or parts of definitions.

As in the normal KREME concept editing views, both inherited and defined features can be displayed.

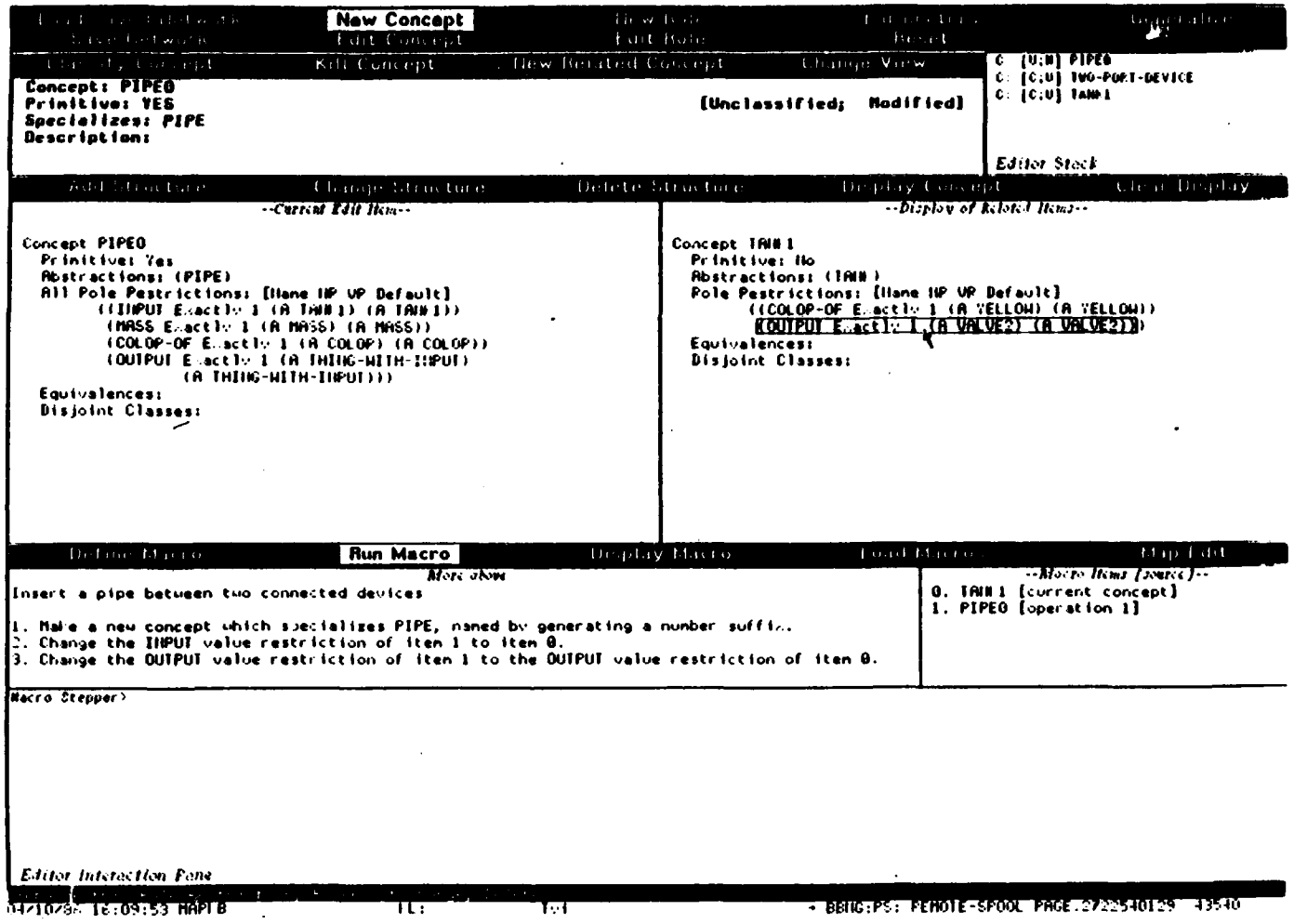


Figure B-17: The Macro Structure Editor View

Clicking the mouse over the keyword indicating each feature class in a concept's definition (e.g., Abstractions:, Slots:, Equivalences:, etc.) toggles the display of that component set of features between *defined* and *all inherited features* of that type.

There is a menu of commands for displaying and editing definitions in these windows. It includes:

- Add Structure - Clicking this command followed by one of the concept feature keywords **Abstractions:**, **Slots:**, **Equivalences:**, **Disjoint Classes:** causes KREME to prompt for a new object of that type. The new item can be typed in or copied from some other visible concept's definition by pointing.
- Change Structure - Clicking this command and then the item to be replaced (a parent, slot name, value restriction, number restriction, default, an equivalence or component path, or a disjoint class) causes KREME to ask for an appropriate replacement. Again the new value may be typed in or pointed to.
- Delete Structure - Clicking this command and then the item to be deleted removes it from the concept's definition.
- Display Structure - Pointing at this command and then any visible concept name or definition places the definition of the concept in the Display Structure Window.
- Clear Display - Removes all definitions from the display window.

Arguments (if any) to these commands may be described by pointing or typing. For example, to delete a slot, click on Delete Structure and the display of the slot to be deleted. To change (that is, replace) a structure, point in succession at the Change Structure command, the item to be replaced, and the thing to replace it with.

In many cases, Delete Structure and Change Structure can also be invoked simply by pointing at the structure to be replaced, and clicking the left mouse button. Delete Structure is often available on the menu of right button options (check the **mouse documentation window**.

Individual concepts can be deleted from the display window by pointing at them and clicking the right button.

The Edit Concept command is used to change what is displayed in the current edit window. Editing a new concept moves the old edit concept to the bottom of the display window.

B.4.3 Developing Macro Editing Procedures

Globally available commands for defining new concepts and specializing old concepts by copying their definitions together with the commands in the Structure Editor's main menu provide an extremely flexible environment in which to define and specify modifications of concepts with respect to other defined concepts. Virtually all knowledge editing operations can be done by a sequence of pointing steps using the current edit window and the display window. This combination of editing features and mouse-based editor interaction style provides an extremely versatile environment for the description, by example, of a large class of editing macros.

The windows on the bottom of the Macro and Structure Editor Screen are used for defining, editing, and running macros composed of structure editing operations.

To define a macro, first edit a concept for which the macro will make sense, and then click on the Define Macro command from the menu below the structure editing windows.

Until the macro definition is terminated by clicking on Define Macro again, all editing and concept display operations performed will be recorded as steps in the macro, and displayed in the lower left window of the screen in English. Specific objects mentioned as arguments will be replaced by references to *macro items*, which are numbered and appear in a list in the lower right window.

B.4.4 Changing features into concepts: A Sample Macro

It is easiest to understand how to use the macro facility by looking at an example.

In developing frame representations, the choice must often be made between defining a slot to denote that the concept has some attribute (e.g., defining RED-CAR as a CAR with slot COLOR-OF restricted to (A RED)), and defining the concept by making it specialize another concept that stands for the class of objects with that attribute (e.g., defining RED-CAR as a CAR and a RED-OBJECT.)

When this choice has been made in a way that later seems awkward or inappropriate, given the use that the concept has in the knowledge-based system under development, it can be very time consuming to change. With KREME, however, macros can be defined that can make the change in either direction.

We illustrate this kind of restructuring operation with a macro that provides a way of forming a concept RED-OBJECT denoting the set of all objects with the role restriction COLOR-OF = RED. The macro makes use of the classifier to find all such classes and make them children of RED-OBJECT, and then remove the COLOR-OF slots from all classes that were found to denote red objects. This macro can be used on all colors defined in the knowledge base, to completely eliminate references to COLOR-OF slots.

The following sequence of steps, all of which were specified, by example, using operations available in the Macro Structure Editing View, accomplishes this task. Figure B-18 shows this macro's steps.

Step 1 creates the concept RED-OBJECT as follows: First, the command New Related Concept was invoked using the *right mouse button* and specifying that the concept OBJECT was to be specialized. The use of right button exposed a set of options on how the object should be named that included adding a prefix to the name of the parent, OBJECT. Clicking on the current editor object RED specifies that the name should be RED-OBJECT and that subsequent uses of the macro on other colors, like GREEN, will create concepts like GREEN-OBJECT.

Next, the COLOR-OF slot of RED-OBJECT was changed to RED by pointing at Change Structure, the old value restriction (A COLOR), and the concept RED.

Steps in COLOR-OBJECTS macro:

Edit Concept RED

Click on Define Macro.

(*Makes Macro Item 0 = RED*).

1. Make a new concept which specializes OBJECT, named by adding as prefix item 0's name (*Creates RED-OBJECT as item 1, puts it in the current edit item window*).
2. Change the COLOR-OF value restriction of item 1 to item 0 (*RED*).
3. Change the primitiveness of item 1 to No.
4. Classify item 1. (*This finds all concepts with COLOR-OF slots restricted to RED, and makes them specializations of RED-OBJECT.*)
   The remaining steps make these specialization links *defined links*, and remove the COLOR-OF slots completely.
5. Do on SPECIALIZATIONS of item 1: Add item 1 to the parents of iteration item. (*This makes each red object have defined parent RED-OBJECT.*)
6. Do on SPECIALIZATIONS of item 1: Classify iteration item.
7. Change the primitiveness of item 1 to Yes.
8. Delete the COLOR-OF restriction of item 1.
9. Do on ALL SPECIALIZATIONS of item 1: Delete the COLOR-OF restriction of iteration item.
10. Classify item 1.

Figure B-18: Changing RED to RED-OBJECT

Step 3 was done by clicking on Primitive: and entering the new value NO. Step 4 was simply the command Classify Concept. So that all red object classes could be found and made specializations of RED-OBJECT.

The remaining steps, required to add defined parents to specializations of RED-OBJECT and to remove their COLOR-OF restrictions, make use of the KREME Structure Editor's Map Edit command. This command is used to perform a single editing operation on a set of concepts related to the one being edited (e.g., direct specializations, all specializations, abstractions, all abstractions). For example, Step 5 was created by the mouse sequence Map Edit, Specializations, RED-OBJECT, Add Structure, the keyword Abstractions: of the specialization that appears temorarily in the edit window, and finally pointing to the concept definition RED-OBJECT.

B.4.4.1 Running Macros

To run the macro on other objects, first edit the concept you wish to start with, then click right on Run Macro and select Current Macro from the pop-up. If you want to do the macro one step at a time, also click Single Step. When you exit this pop-up menu, another will appear from you will be asked to select which sets of relatives of that concept (Specializations, All Specializations, etc.) you wish to run the macro on.

Individuals only means only apply the macro to concepts that are marked as individuals. **Include current concept** asks if you wish to run the macro only on the relatives, and not the current concept itself.

If you use the single stepper, then you will interact with the **Macro Stepper>**, which has the following commands:

- **Help** - print the list of commands.
- **Execute** the next step in the macro.
- **Proceed** with the rest of the steps without stopping.
- **Skip** execution of the next step.
- **Delete** the current step from the macro.
- **Insert** a step into the macro at this point. (which you specify)
- **Quit** the macro.

To load previously saved macros, use the Load Macros command. A pop-up menu will display the files that contain saved macros. (The macro file for coloring objects is in the file COLOR-OBJECT.)

To display a loaded macro, use the Display Macro command. This command also makes a loaded macro the current one.

To save a macro, use the Save Macro command from the menu on the name of the macro displayed in the macro definition window.

B.5 The Generalizer

Experienced knowledge engineers are often able to discover and define useful generalizations that help organize the knowledge described by a human domain expert. The expert, although not previously aware of such a generalization, will often immediately perceive its relevance to his own reasoning processes, going so far as to suggest improvements, related generalizations, more abstract generalizations and so forth.

As an initial experiment in automatic generalization within frame taxonomies, KREME provides a relatively simple generalizer algorithm that relies on the user to select from a set of potential generalizations discovered essentially by exhaustive search.

To use the generalizer, click on Generalize in the main menu. KREME will then start a background process[24] to search for pairs or larger sets of concepts that share some number of features (slots, equivalences,

[24]Because the generalizer algorithms is fairly slow (taking about 8 minutes to go through a network of 500 concepts and 300 roles), it runs as a low priority background process, looking for generalization only when the editor is waiting for input from the user.

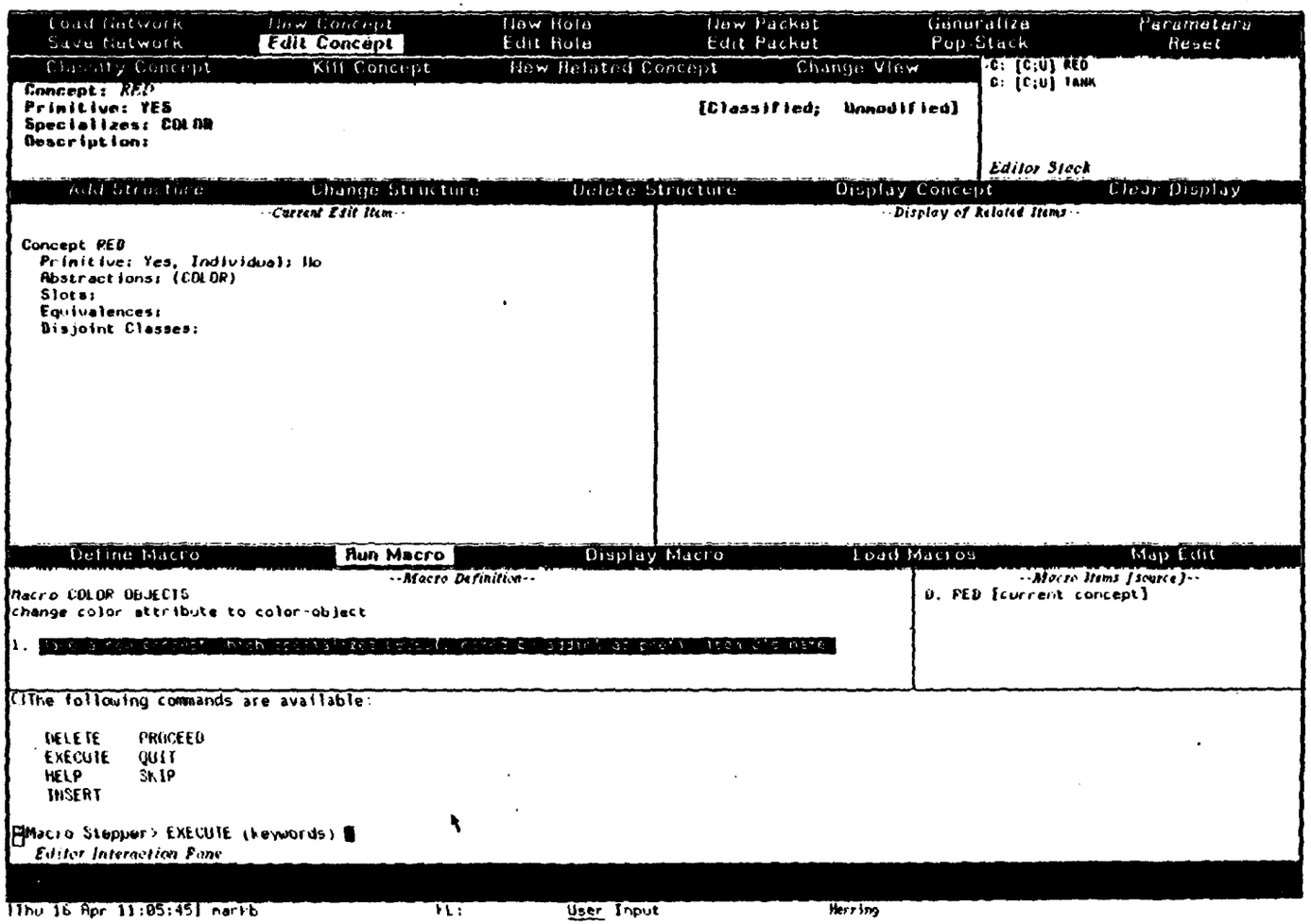


Figure B-19: Running the macro COLOR-OBJECT

parents, etc.) For each such set it finds, the generalizer will then form the most specific concept definition that encloses all of the features but is more general than any concept in the set. This concept definition, a potential new abstraction of a number of concepts, will be displayed to you. If you find that the generalization is useful, specify a name when prompted. The newly named concept is then classified and inserted into the network.

To run the Generalizer:

- Click on Generalize
- The Generalize Command will be highlighted and will remain highlighted until it finds a generalization. At that time the Generalize will blink to alert you that it has found a generalization.
- Hit <SUPER> <REFRESH> to make KREME show you what it has found. You will see a menu of choices prompting you to make and clarify the concept:
  - Y to reject the concept.
  - N to defer making your choice until you have more information. Deferring will pop you back to the state of the network before you typed <SUPER> <REFRESH>.
  - D to form the concept without classifying it, making it the top item on the Edit stack. KREME will ask you to give the new concept a name.
  - E to Edit the definition of the new generalization.
- Click on <SUPER> <ABORT> to end generalization.

B.6 The KREME Rule Editor

B.6.1 Introduction

In Expert Systems, rules are often organized into packets and the requirements for altering and inspecting the relationships between rules have analogs in the packet domain. KREME provides facilities to see displays of the relationships between packets, and to inspect the internal structure of packets and rules.

KREME's Rule Editor is equipped with a number of features that facilitate building and maintaining knowledge bases of rules and rule packets. The Rule Editor uses the same basic operations as the Macro Structure editor discussed earlier. It contains facilities for creating and editing rules and rule packets, copying rules, moving them, compiling rules and displaying and modifying variable bindings. The system provides an elementary history and tracing mechanism, and an explanation system that produces pseudo-English explanations from rule traces.

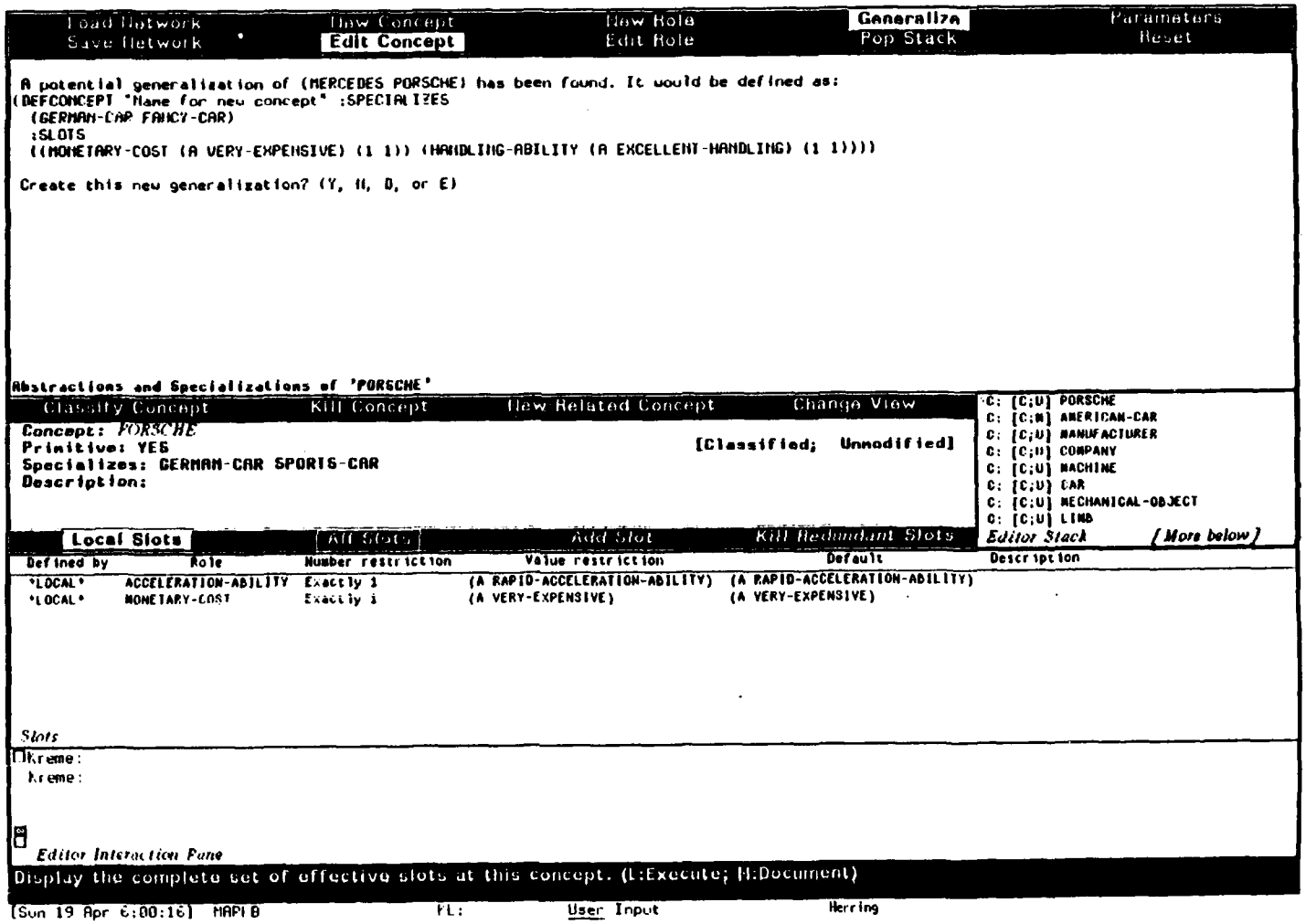


Figure B-20: Finding a new generalization.

B.6.2 Editing Rules in the KREME Environment

KREME at present edits rules in the FLEX [17] rule language. In FLEX, rules come in *rule packets*, and the KREME Rule Editor edits an entire packet at one time. Rule packets provide a way to organize rules.

The forward chaining rule packets come in four varieties, indicating the type of control mechanism used for rule firing.

- **do-1-rule-packets** execute the first rule whose test succeeds.
- **do-all-rule-packets** execute all rules whose tests succeed.
- **while-1-rule-packets** repeatedly test all rules, firing one, until no tests succeed.
- **while-all-rule-packets** repeatedly fires all rules whose tests succeed, until none succeed.

Rule packets are connected to KREME frame systems or other data contexts by specifying an *access environment*. An access environment is an object that receives messages dealing with the accessing of values for references in the rules. It handles all messages to get or set the values of variables and their confidences.

B.6.3 The KREME Rule Editor

Rules are defined and edited by specifying and filling out portions of rule *templates*. To refine these templates either use the mouse to copy parts of existing rules or point at slots to be filled and type in the desired values.

There are also commands to run packets and debug them and to generate traces or rule histories paraphrased in pseudo-English, and delete rules and reorder rules, and load and save rules from files.

B.6.4 The Rule Editor View

Many of the windows in the **Rule Editor View** should be familiar by now. The complete list is as follows:

1. **Global Command Window** displays global commands that can be selected by the user. In this example, the user has used the mouse to select Edit Packet. The user's selection is highlighted.
2. **State Window** displays the name of the packet, the network it is associated with, and other useful information.
3. **Editor Stack Window** displays the names of the items recently edited and some information on their current state. Items in the editor stack window can be selected for editing with the mouse.
4. **Behavior Command Window** is a menu of commands that apply to Rules and Rule Packets. (*Behavior* is another term for rule packets, or functional methods on instances of concepts.)
5. **Current Edit Item Window** displays the item that has been selected for editing.
6. **Dispay Related Items Window** allows the user to view other rule packets and scroll through them. Rules and parts of rules can be copied from the Scroll Window into the Current Edit Item Window.
7. **Editor Interaction Window** displays screen prompts and user input. The user's edits are made in this window and then displayed in the Current Edit Item Window.

8. **Related Behaviors Window** displays an index of other rule packets that are related to the one currently being edited. With the mouse, the user can rapidly scroll through this index and select a *related rule packet* for viewing or editing.

To get into the Rule editor use the [New Packet] or [Edit Packet] command in the **global command window**.

Thereafter, you can use the structure editor in much the same way the Macro Structure Editor is used to edit concepts. The **Rule Structure Command Menu** contains the commands:

- [Define Behavior] is similar to [Classify Concept]. It makes the definition of the packet permanent, and allows it to be run or attached to a concept.
- [Similar Behavior] - Creates a packet with the same rules, etc. but gives it a new name, and presents it to be edited to make it different.
- [Kill Behavior] - Kills the definition of this packet.
- [Display Packet] - Displays the packet in the **Display of Related Items Window**.

When a whole rule packet is outlined (such as when you are over the word **Packet**), you can choose to [Edit Packet] (L:), or (R:) choose from a menu of [Edit Packet], [Edit Basis] or [Display Lisp Form].

Other editing commands are found on the keywords and component pieces of packets and rules. For instance, clicking left on [Rule:] places a new (empty) rule in the packet, which can then be filled out by clicking on [IF] to add a new condition (conditions are treated as part of a conjunction) or [THEN] to add a new action. Clicking right gives a menu of [Add (Empty] Rule), [Copy One Rule] from somewhere else into this packet, and [Copy Rule Set] which copies all of the rules from another packet.

Clicking over [Type:] gives you a choice of the standard types of rule packets, described above.

[Packet Classes:] allows you to specify a flavor to be mixed into the packet. [Arguments:] and [Return Variables:] each allow you to add a new one (L:) or choose from a menu of [Add One], [Add Several], [Edit] and [Replace].

When a whole rule is outlined, clicking left will be replace the rule with another rule that you point at. Clicking right gives a menu of [Replace Rule], [Edit Attributes] and [Delete Rule].

Whenever expressions appear (after the word **Precondition:**, or as parts of conditions or actions), the user may [Replace] the expression (L:), or choosing from a menu (R:) of

- [Replace] the expression with another one.
- [Edit] the expression as text.
- [Delete] the expression.

- **Add Before** another expression (copied from somewhere by pointing).
- **Add After** another expression.
- **Exchange** two expressions positions.
- **Parenthesize** a set of expressions together.
- **Deparenthesize** an expression into pieces.
- **Evaluate** the expression in the current context.

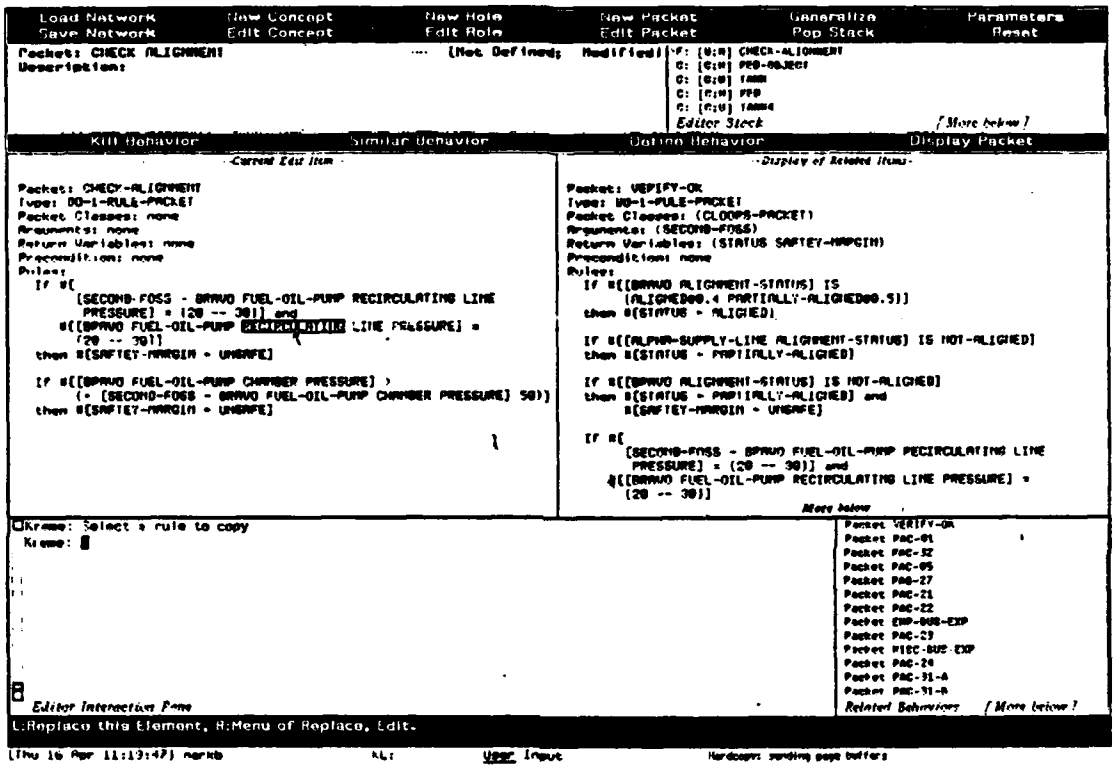


Figure B-21: The KREME Rule Editor

Appendix C
A Session With KREME

This appendix shows screen hardcopies of an example user session.

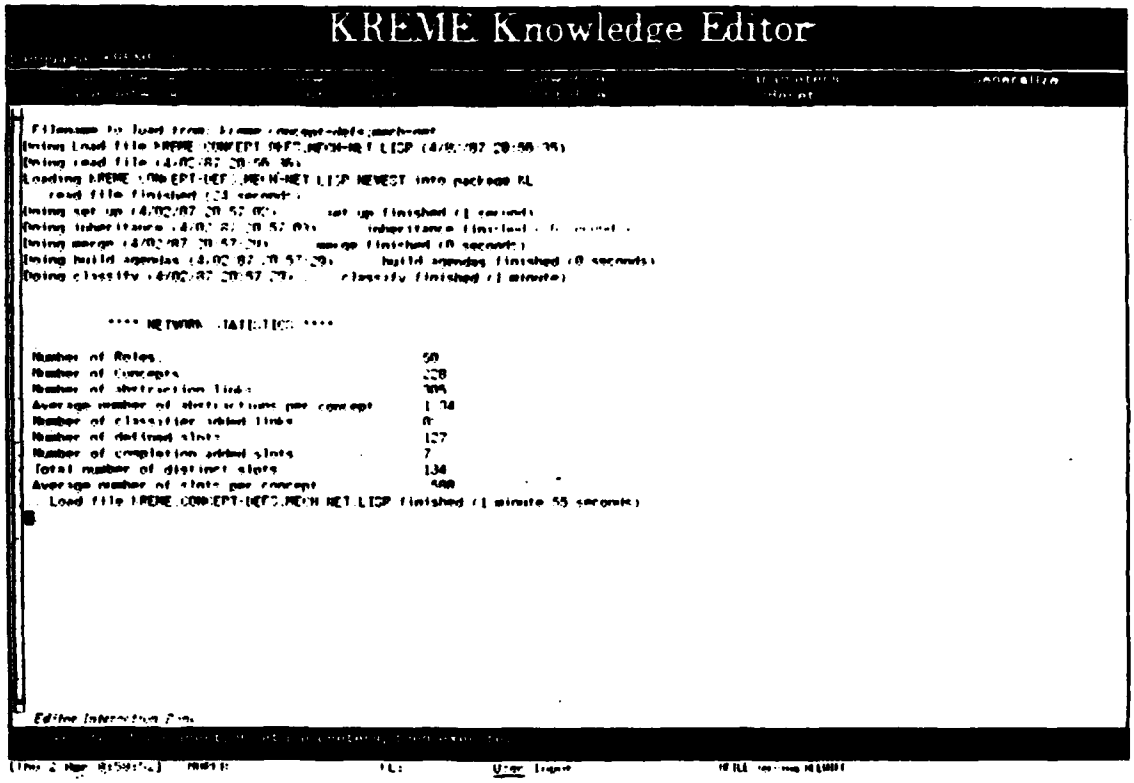


Figure C-1:

This is the initial state of the KREME interface. To reach this window, the user logged on and typed <select> K. Here the user has clicked on Load Network and typed the name of the file containing the network he wishes to load. The file in this particular case is named KREME:CONCEPT-DEFS;MECH-NET.LISP. KREME displays information about the loaded network as it reads it.

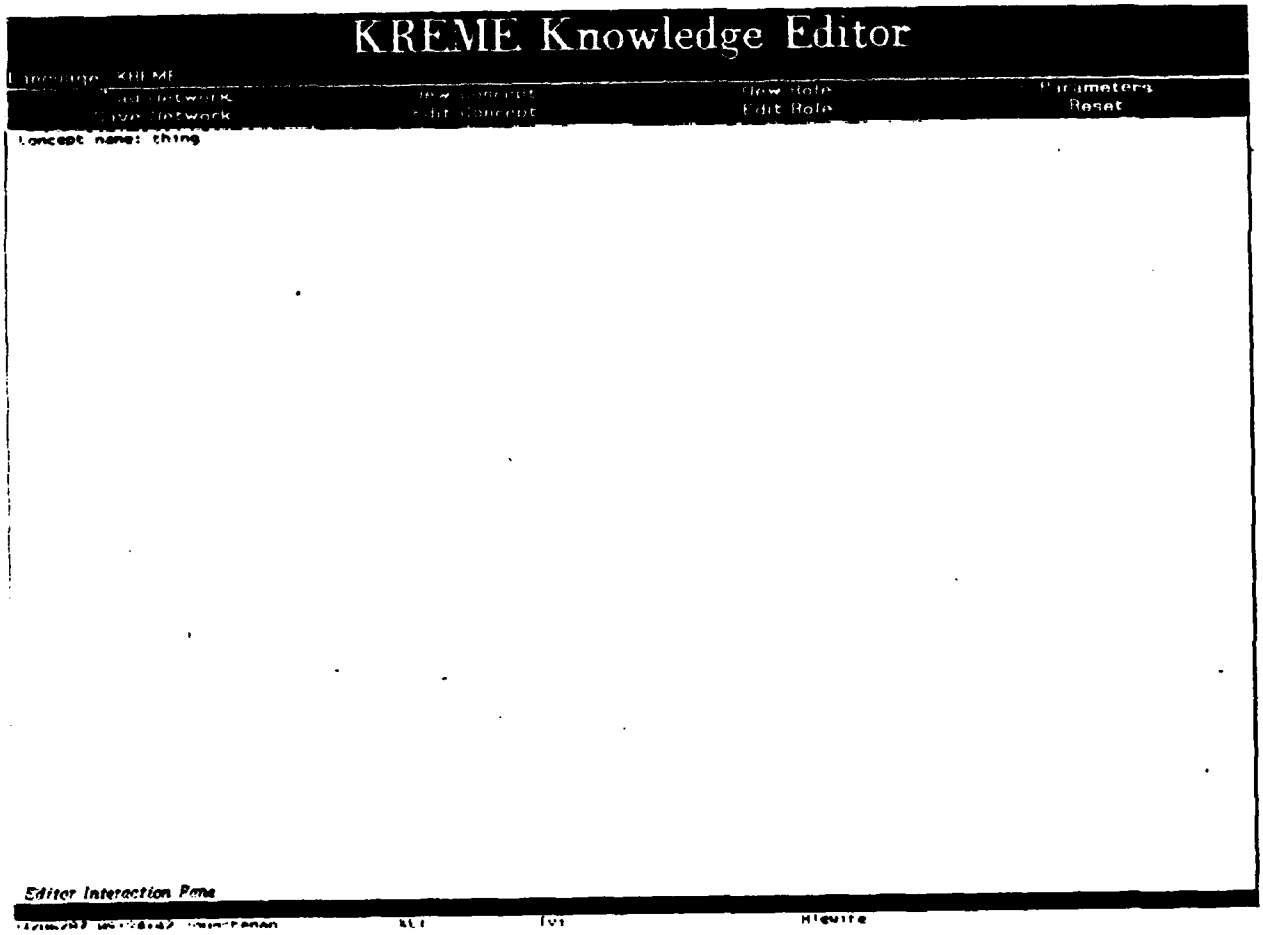


Figure C-2:

Next, the user clicked on Edit Concept and typed "thing" to the prompt.

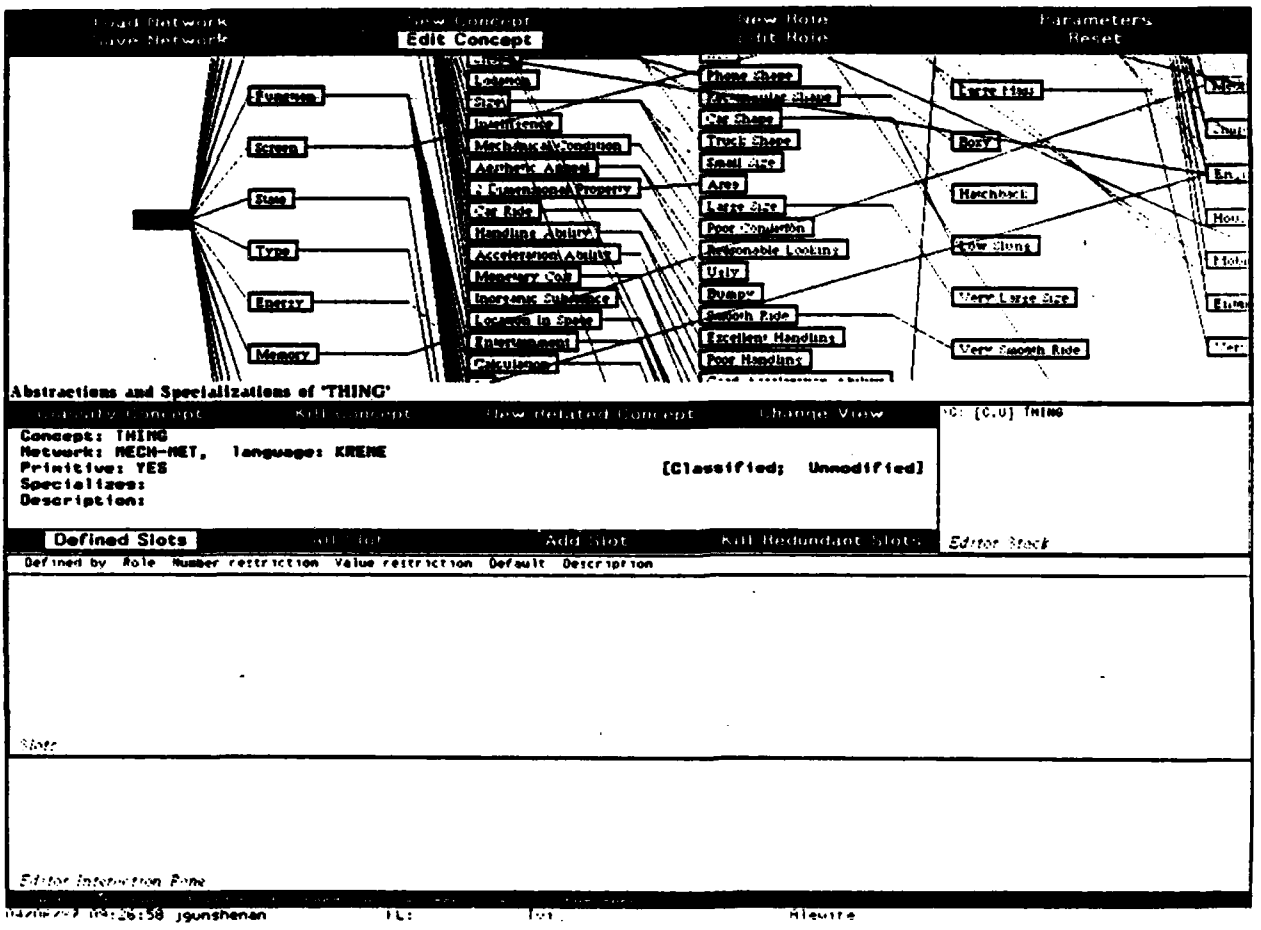


Figure C-3:
This is the "Main Concept View" of the network, showing the top frame in the network, THING.

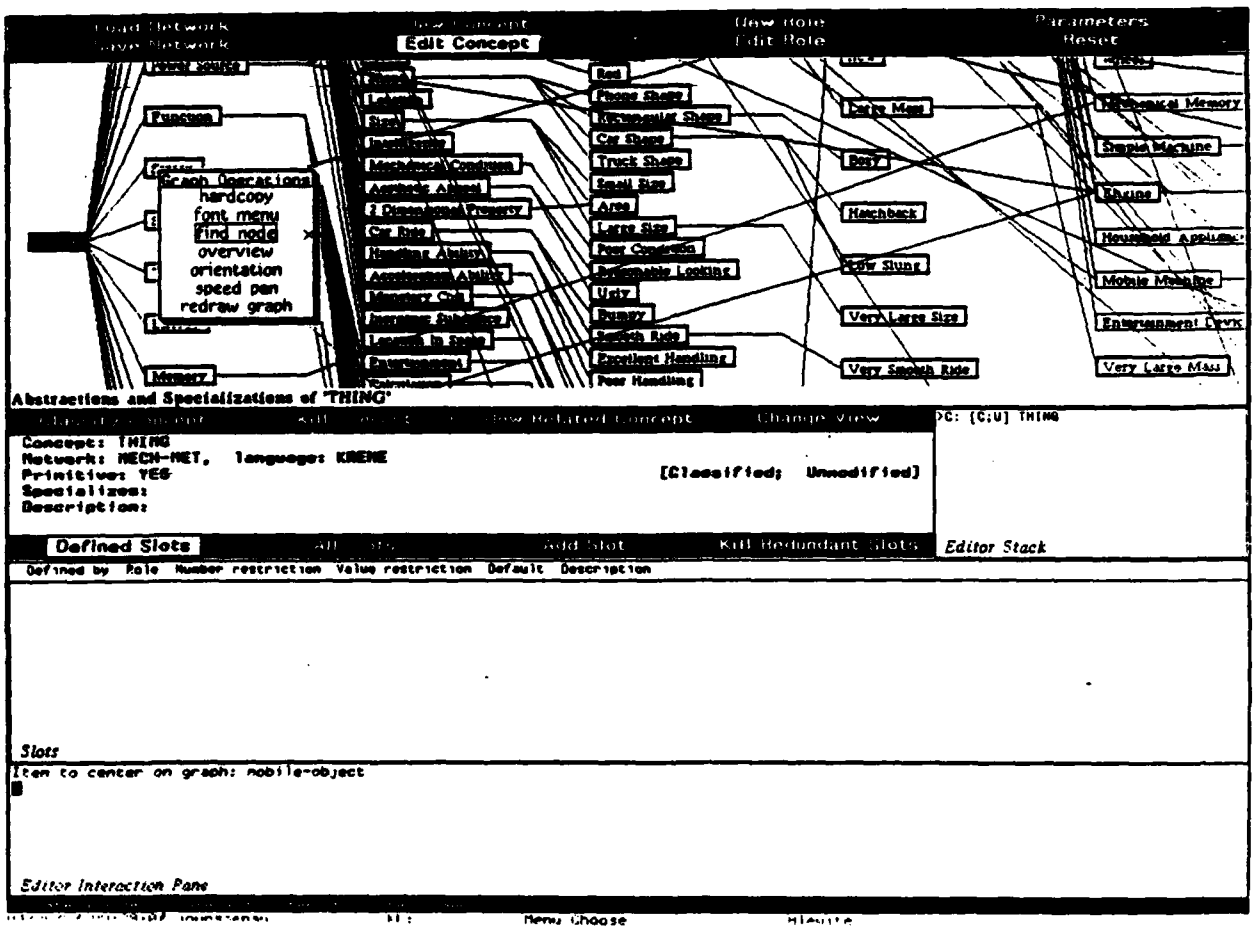


Figure C-4:

Here the user clicked the right mouse button while pointing to an empty spot in the graph window. This menu displays operations that can be performed on the network display. The user selected find node and typed the concept name "mobile-object" to the prompt.[25]

[25] In names with hyphens, the grapher replaces these hyphens with spaces in the display. For example, the grapher displays the concept "mobile-object" as "Mobile Object". The user must type the hyphen whenever referring to such concepts.

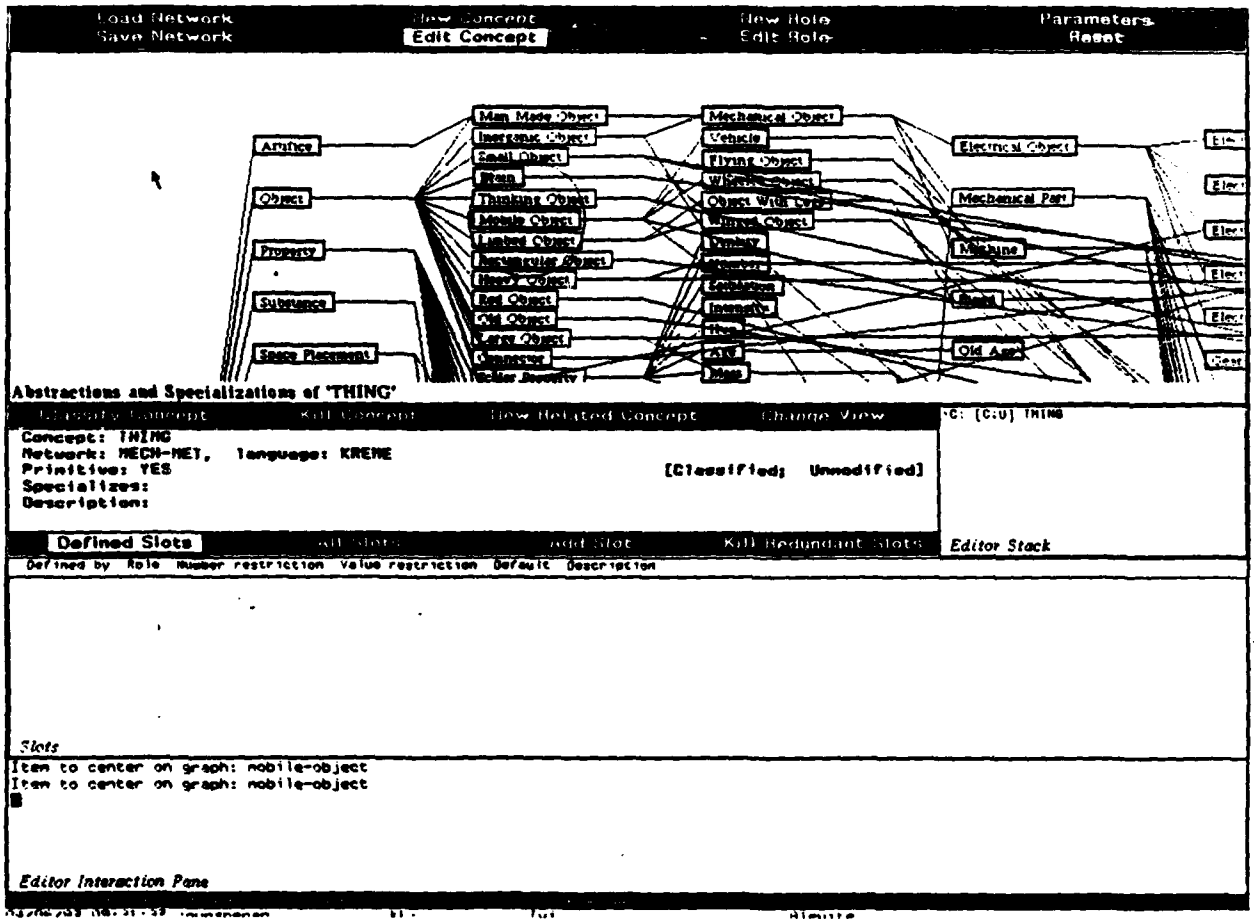


Figure C-5:

The **find node** command circles the node and centers it (roughly) on the display.

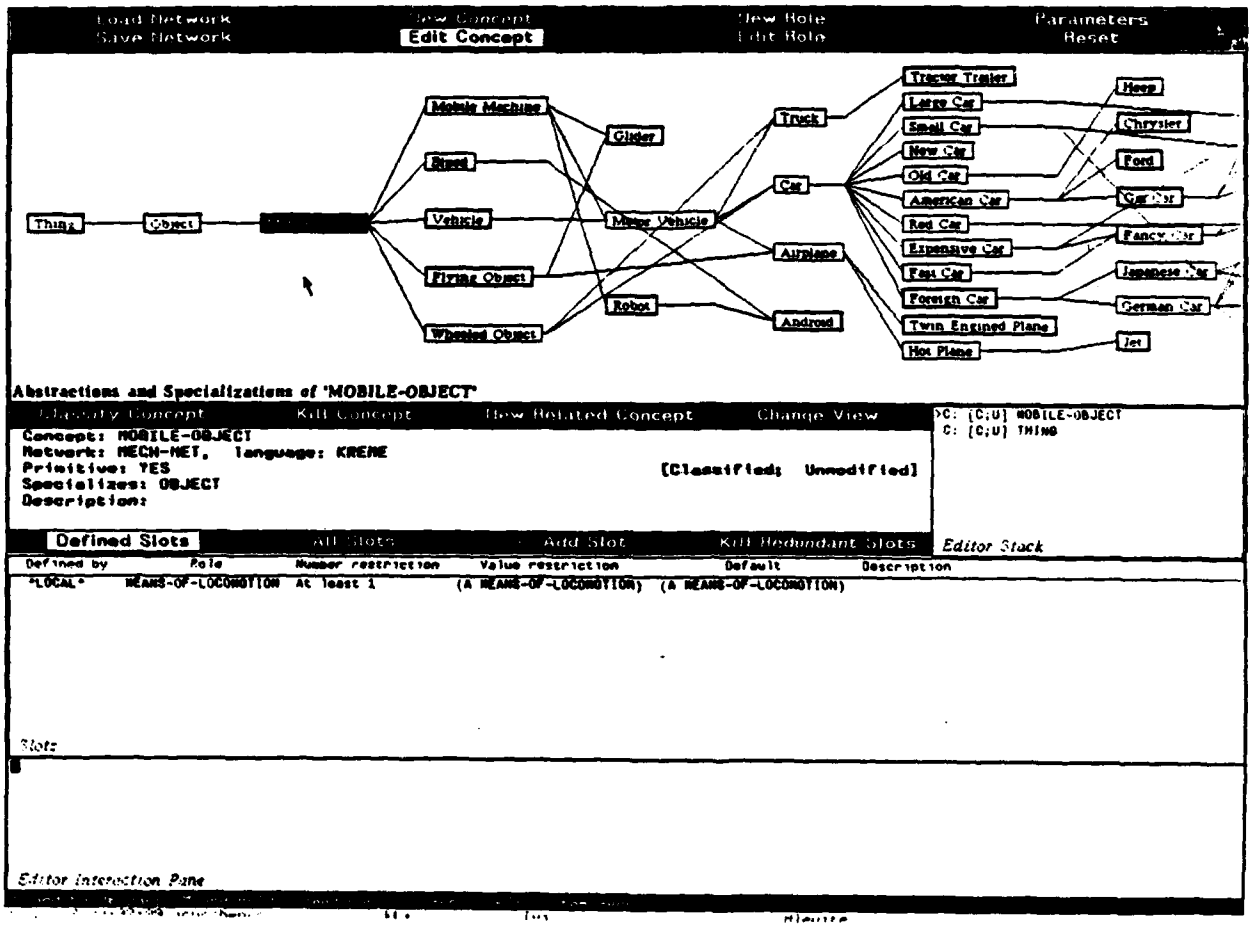


Figure C-6:

Here the user has clicked the left button over the node for the concept Mobile Object, causing that to be the current editor object. The graph now displays only the concepts that are abstractions (parents, parents of parents, etc.) and specializations (children, children of children, etc.) of MOBILE-OBJECT. In the table editing window, the locally defined slots of MOBILE-OBJECT are displayed.

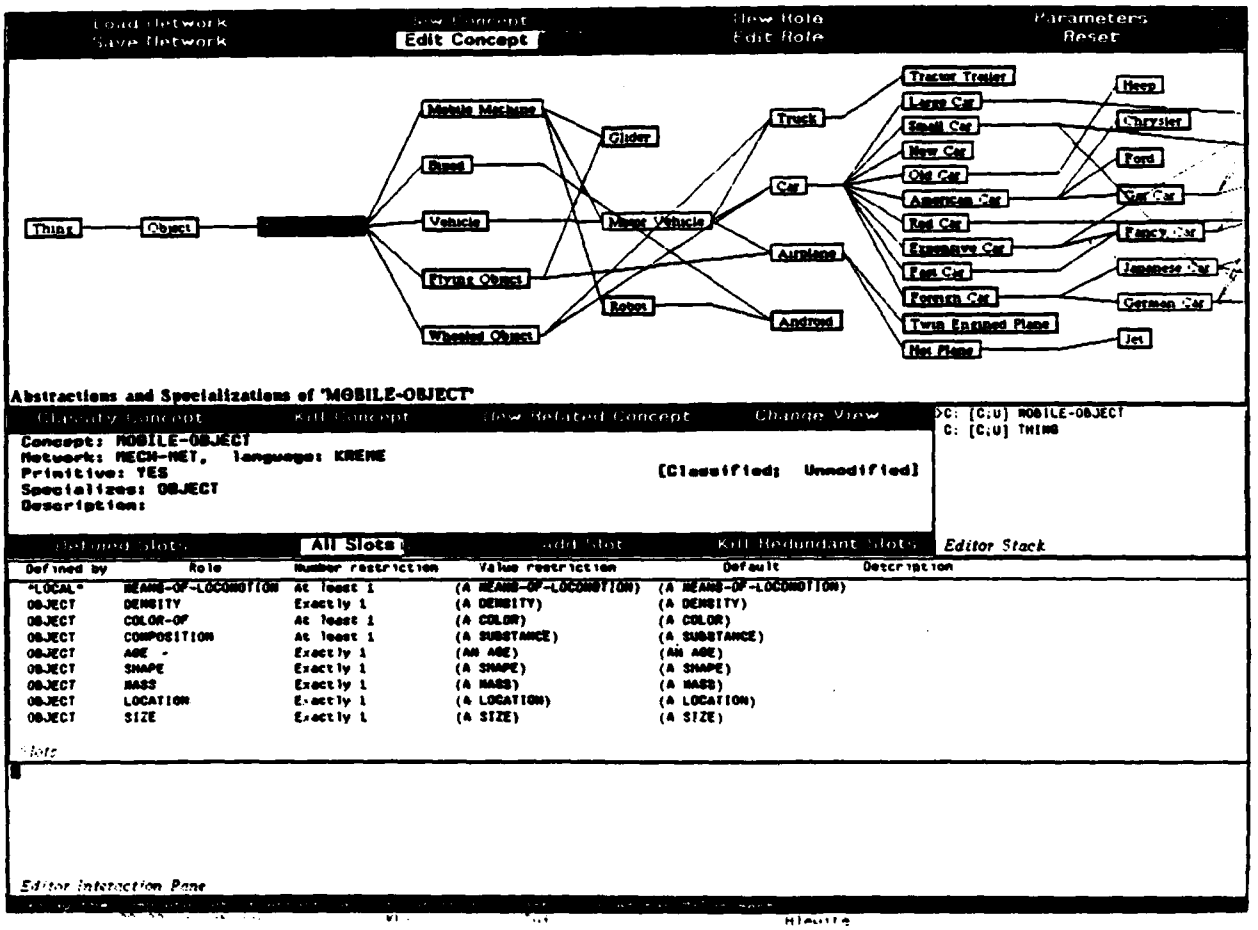


Figure C-7:

Here the user has clicked on **All Slots**. KREME now displays all the inherited slots in addition to the slots local to the concept. The table shows where the slot was inherited from in the first column, "Defined By".

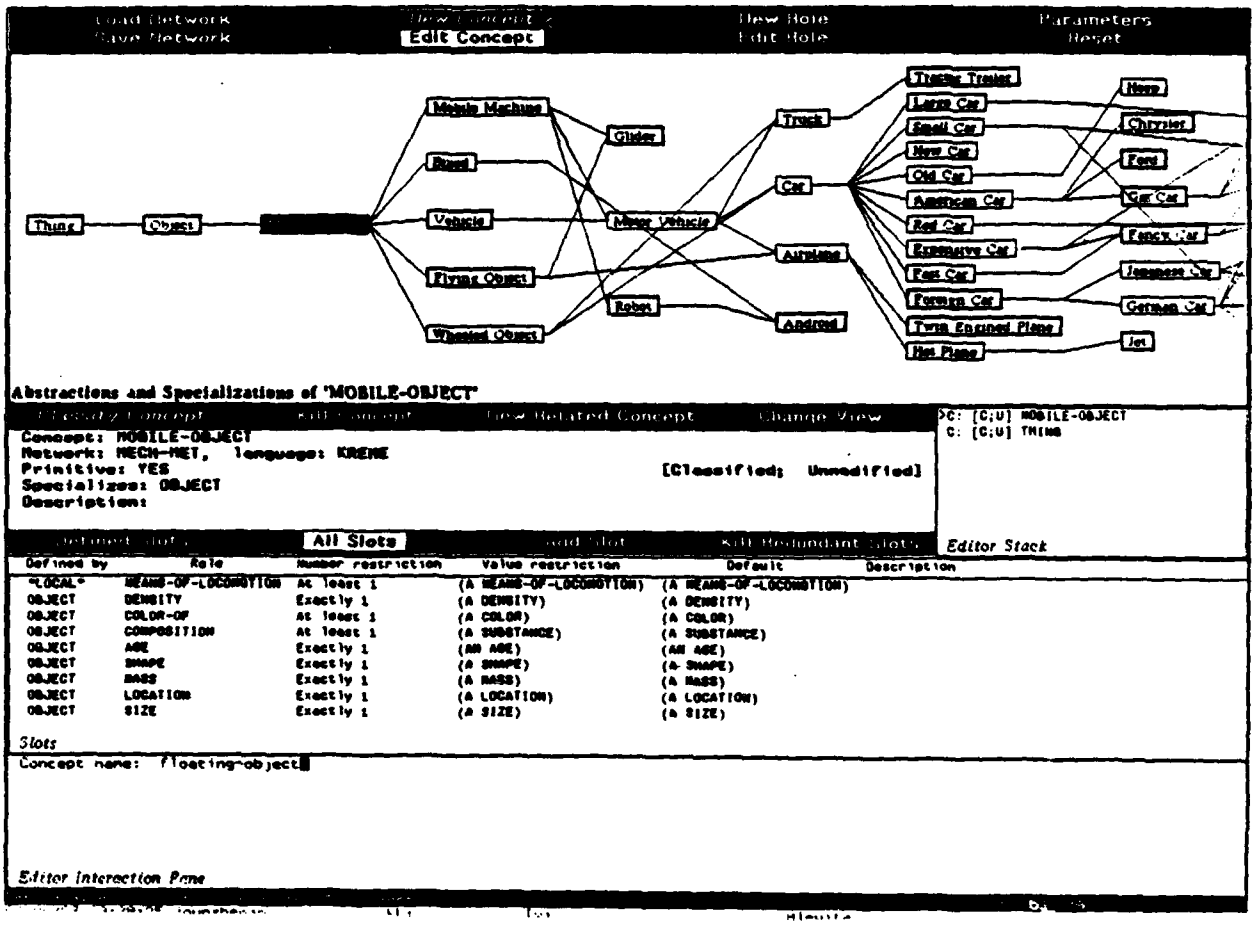


Figure C-8:

Here the user is adding a new concept. After clicking New Concept, the user gave it the name "FLOATING-*OBJECT*".

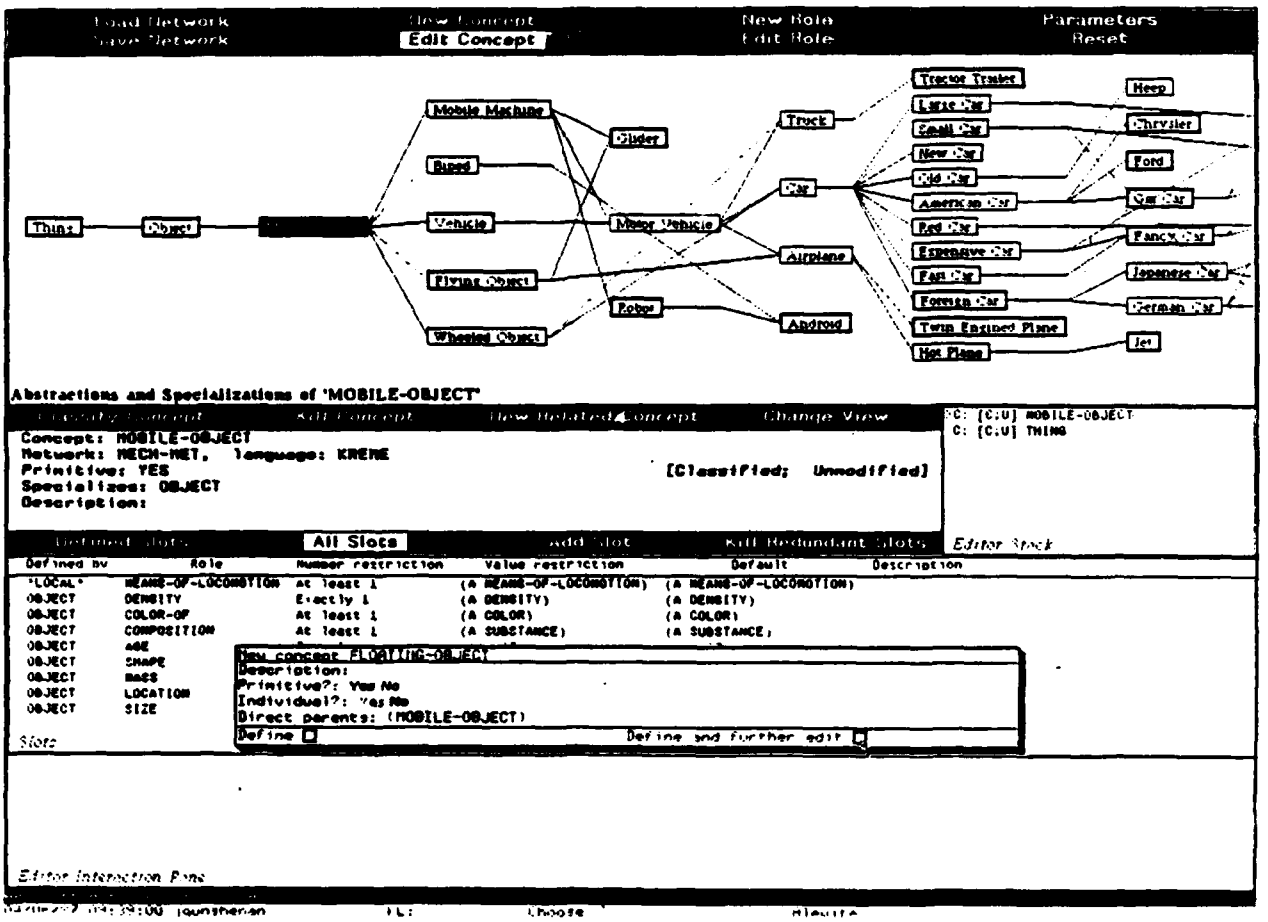


Figure C-9:

KREME pops-up a menu containing the basic things it needs to know about a new concept. The user changes the **Direct Parents** field from the default (THING) to (MOBILE-OBJECT). Note the hyphen. **Direct Parents** is a list (in parentheses) of the concepts that the new one specializes. Next, the user clicks on the [x] **Define and further edit** box. The pop-up menu disappears, and the new concept becomes the current editor object.

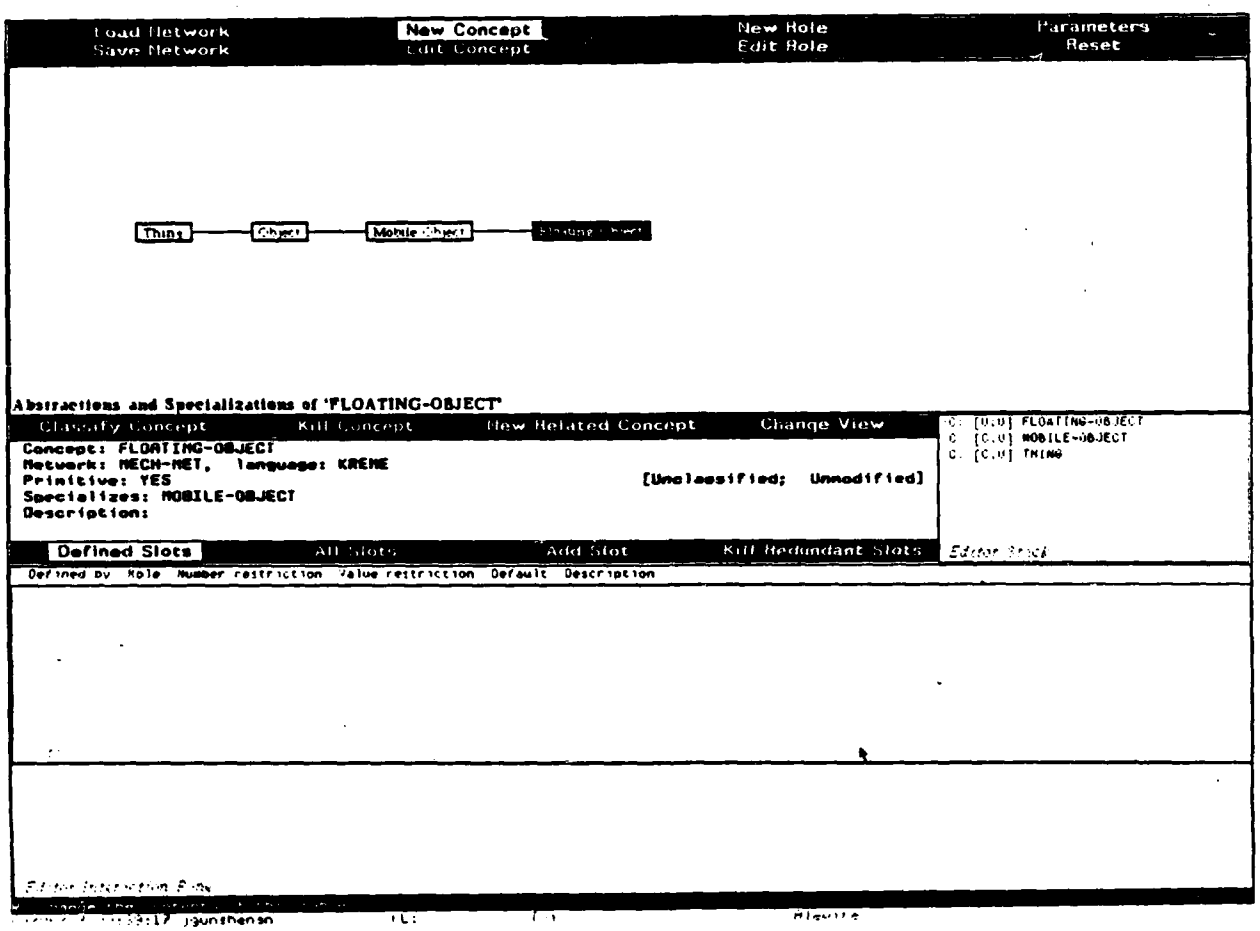


Figure C-10:

The new concept, FLOATING-OBJECT, is available at this stage for further definition and editing. Common operations would be to look at inherited slots, add new ones, enter a description, etc. Finally, the definition must be classified with the Classify Concept command before it becomes a permanent part of the network.

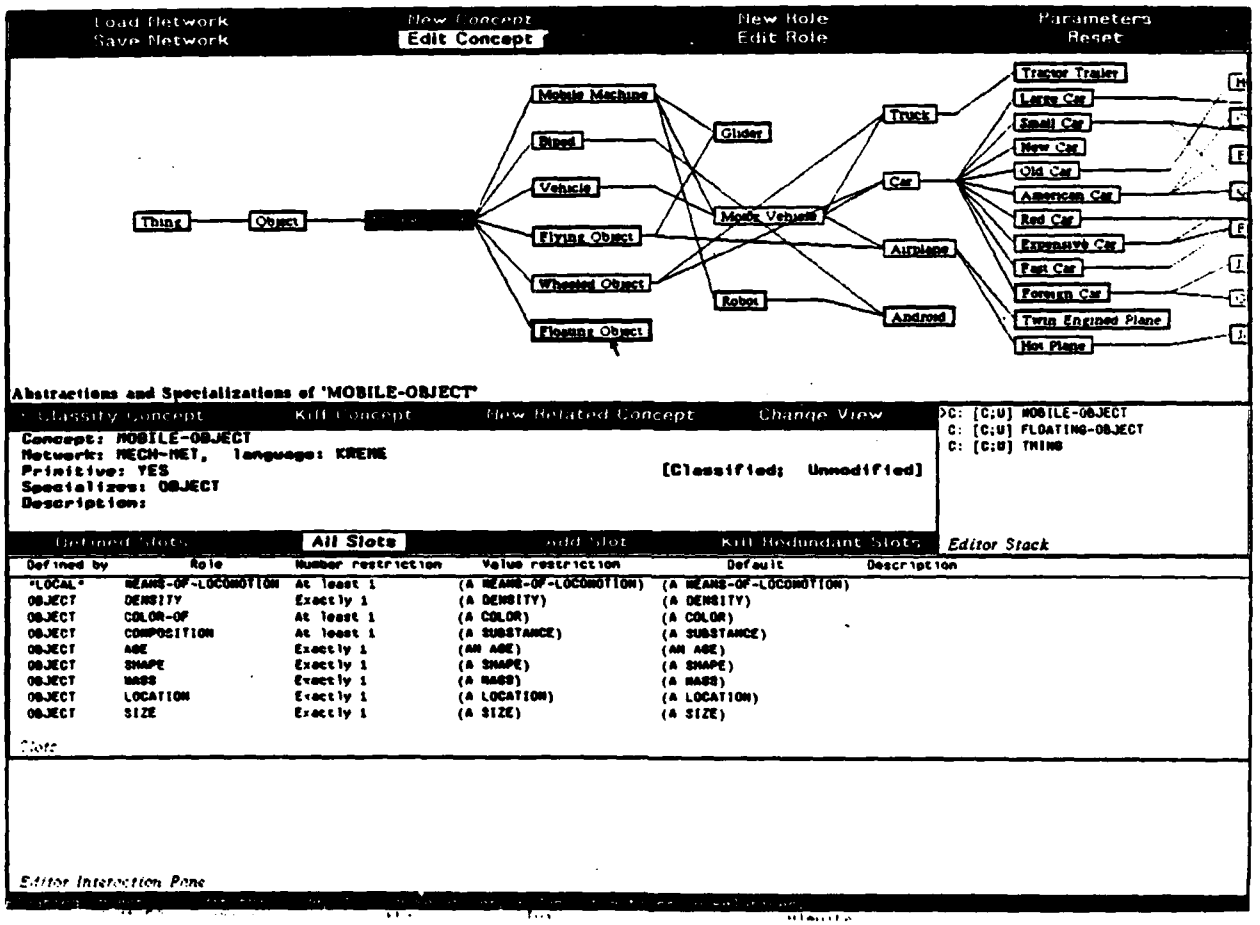


Figure C-11:

By clicking the left button over MOBILE-OBJECT in the Editor Stack Window, or by using the Edit Concept command again, KREME returned to a view of the concept "MOBILE-OBJECT". The graph now shows the new child concept "FLOATING OBJECT".

Bibliography

[1] Balzac, Stephen R.
A System for the Interactive Classification of Knowledge.
Technical Report M.S. Thesis, M.I.T. Dept of E.E. and C.S., 1986.

[2] Bobrow, D., Winograd, T. and KRL Research Group.
Experience with KRL-0: One cycle of a knowledge representation language.
In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*. IJCAI-77, Cambridge, MA., August, 1977.

[3] Brachman, R.J., Fikes, R.E., and Levesque, H.J.
Krypton: A Functional Approach to Knowledge Representation.
IEEE Computer, Special Issue on Knowledge Representation , October, 1983.

[4] Carbonell, Jaime G.
Derivational Analogy: A theory of reconstructive problem solving and expertise acquisition.
In Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (editor), *Machine Learning: Volume II*, pages 371-392. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[5] Ernst, G.W. and Newell, A.
GPS: A Case Study in Generality and Problem Solving.
Academic Press, New York, 1969.

[6] IntelliCorp.
KEE Software Development System.
IntelliCorp, 1984.

[7] Keene, Sonya E. and Moon, David.
Flavors: Object-oriented Programming on Symbolics Computers.
Symbolics, Inc.
1985

[8] Carnegie Group, Inc.
KnowledgeCraft.
Carnegie Group, Inc., 1985.

[9] Moser, Margaret.
An Overview of NIKL.
Technical Report Section of BBN Report No. 5421, Bolt Beranek and Newman Inc., 1983.

[10] Newell, A.
The knowledge level.
AI Magazine 2(2):1-20, 1981.

[11] Rich, C.
Knowledge Representation Languages and Predicate Calculus: How to Have Your Cake and Eat It Too.
In *Proc. AAAI*, pages 192-196. 1982.

[12] Roberts, B. and Goldstein, I. P.
The FRL Manual.
A.I. Lab. Memo 409, M.I.T., 1977.

[13] Sacerdoti, E.
Planning in a Hierarchy of Abstraction Spaces.
Artificial Intelligence 5(2):115-135, 1974.

[14] Sacerdoti, E.
A structure for plans and behavior.
Technical Report 109, SRI Artificial Intelligence Center, 1975.

[15] Schmolze, J. and Israel, D.
KL-ONE: Semantics and Classification.
In *Research in Knowlege Representation for Natural Language Understanding, Annual Report: 1 September 1982 to 31 August 1983.* BBN Report No. 5421, 1983.

[16] Schmolze, J.G., Lipkis, T.A.
Classification in the KL-ONE Knowledge Representation System.
In *Proc. 8th IJCAI.* 1983.

[17] Shapiro, Richard.
FLEX: A Tool for Rule-based Programming.
Technical Report 5643, BBN Labs., 1984.

[18] Sidner, C.L.; Bates, M.; Bobrow, R.J.; Brachman, R.J.; Cohen, P.R.; Israel, D.J.; Webber, B.L.; and Woods, W.A.
Research in Knowledge Representation for Natural Language Understanding: Annual Report.
Technical Report BBN Report No. 4785, Bolt Beranek and Newman Inc., November, 1981.

[19] Stefik, Mark.
Planning with Constraints: MOLGEN.
Artificial Intelligence 16(2):111-169, 1981.

[20] van Melle, W.
A domain independent production-rule system for consultation programs.
In *Proceedings of IJCAI-6*, pages 923-925. August, 1979.

[21] Vilain, Marc.
The Restricted Language Architecture of a Hybrid Representation System.
In *Proceedings, IJCAI-85*, pages 547-551. International Joint Conferences on Artificial Intelligence, Inc., August, 1985.

[22] Williams, M., Hollan, J., and Stevens, A.
An Overview of STEAMER: An Advanced Computer-Assisted Instruction System for Propulsion Engineering.
Behavior Research Methods and Instrumentation 14:85-90, 1981.

DISTRIBUTION LIST

addresses	number of copies
Sharon M. Walter RADC/COES	70
RADC/COVL GRIFFISS AFB NY 13441	1
RADC/DAP GRIFFISS AFB NY 13441	2
ADMINISTRATOR DEF TECH INF CTR ATTN: DTIC-DDA CAMERON STA BG 5 ALEXANDRIA VA 22304-6145	5
RADC/COTD BLDG 3, ROOM 14 GRIFFISS AFB NY 13441-5700	1
Director DMAAC (Attn: RE) 3200 S. Second St. St Louis MO 63118-3399	1
AFCSA/SAMI Attn: Miss Griffin 10363 Pentagon Wash DC 20330-5425	1
HQ USAF/SCTT Pentagon Wash DC 20330-5190	1
SAF/AQSC Pentagon 4D-267 Wash DC 20330-1000	1

DIRECTOR
DMAHTC
ATTN: SCSIM
Wash DC 20315-0030 1

Director, Info Systems
CASD (C3I)
Rm 3E187
Pentagn
Wash DC 20301-3040 1

Fleet Analysis Center
Attn: GIDEP Operations Center
Code 30G1 (E. Richards)
Corona CA 91720 1

HQ AFSC/DLAE
ANDREWS AFB DC 20334-5000 1

HQ AFSC/XRT
Andrews AFB MD 20334-5000 1

HQ AFSC/XRK
ANDREWS AFB MD 20334-500 1

HQ SAC/SCPT
OFFUTT AFB NE 68113-5001 1

HQ ESC/DCOL
Attn: Fred Lacwig
San Artcrio TX 78243-500C 1

DTESA/RQEE
ATTN: LARRY G.MCMANUS
2501 YALE STREET SE
Airport Plaza, Suite 102
ALBUQUERQUE NM 87106 1

HQ TAC/DRIY
Attn: Mr. Westerman
Langley AFB VA 23665-5001 1

HQ TAC/DCA
LANGLEY AFB VA 23665-5001 1

HQ TAC/DRCC
LANGLEY AFB VA 23665-5001 1

HQ TAC/DRCA
LANGLEY AFB VA 23665-5001 1

ASD/ENEMS
Wright-Patterson AFB OH 45433-6503 2

ASD-AFALC/AXS
WRIGHT-PATTERSON AFB OH 45433 1

ASD/AFALC/AXAE
Attn: W. H. Dungey
Wriight-Patterson AFB OH 45433-6533 1

ASD/ENAMA
Wright-Patterson AFB OH 45433-6503 1

ASD/ENAMA
Wright-Patterson AFB OH 45433 1

AFIT/LDEE 1
BUILDING 640, AREA B
WRIGHT-PATTERSON AFB OH 45433-6583

AFWAL/MLPO 1
WRIGHT-PATTERSON AFB OH 45433-6533

AFWAL/MLTE 1
WRIGHT-PATTERSON AFB OH 45433

AFWAL/FIES/SURVIAC 1
WRIGHT-PATTERSON AFB OH 45433

AAMRL/HE 1
WRIGHT-PATTERSON AFB OH 45433-6573

Air Force Human Resources Laboratory 1
Technical Documents Center
AFHRL/LRS-TDC
Wright-Patterson AFB OH 45433

2750 ABW/SSLT 1
Bldg 262
Post 11S
Wright-Patterson AFB OH 454433

AFHRL/OTS 1
WILLIAMS AFB AZ 85240-6457

AUL/LSE 1
MAXWELL AFB AL 36112-5564

HQ AFSPACECOM/XPYS
ATTN: DR. WILLIAM R. MATOUSH
PETERSON AFB CO 80914-5001 1

3280TTG/EISS
Attn: TSgt Kirk
Lackland AFB TX 78236 1

HQ Air Training Command
TTOI
Randolph AFB TX 78150-5001 1

Defense Communications Engineering Ctr
Technical Library
1860 Wiehle Avenue
Reston VA 22090-5500 1

COMMAND CONTROL AND COMMUNICATIONS DIV
DEVELOPMENT CENTER
MARINE CORPS DEVELOPMENT & EDUCATION COMMAND
ATTN: CODE DICA
QUANTICO VA 22134-5080 2

AFLMC/LGY
ATTN: CH, SYS ENGR DIV
GUNTER AFS AL 36114 1

U.S. Army Strategic Defense Command
Attn: DASD-H-MPL
P.O. Box 1500
Huntsville AL 35807-3801 1

COMMANDING OFFICER
NAVAL AVIONICS CENTER
LIBRARY - D/765
INDIANAPOLIS IN 46219-2189 1

COMMANDING OFFICER
NAVAL TRAINING SYSTEMS CENTER
TECHNICAL INFORMATION CENTER
BUILDING 2068
ORLANDO FL 32813-7100 1

COMMANDER 1
NAVAL OCEAN SYSTEMS CENTER
ATTN: TECHNICAL LIBRARY, CODE 964?B
SAN DIEGO CA 92152-5000

COMMANDER (CODE 3433) 1
ATTN: TECHNICAL LIBRARY
NAVAL WEAPONS CENTER
CHINA LAKE, CALIFORNIA 93555-6001

SUPERINTENDENT (CODE 1424) 1
NAVAL POST GRADUATE SCHOOL
MONTEREY CA 93943-5000

COMMANDING OFFICER 2
NAVAL RESEARCH LABORATORY
ATTN: CODE 2627
WASHINGTON DC 20375-5000

SPACE & NAVAL WARFARE SYSTEMS COMMAND 1
PMW 153-3DP
ATTN: R. SAVARESE
WASHINGTON DC 20363-5100

CDR, U.S. ARMY MISSILE COMMAND 2
REDSTONE SCIENTIFIC INFORMATION CENTER
ATTN: AMSMI-RD-CS-R (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5241

Advisory Group on Electron Devices 2
Hammond John/Technical Info Coordinator
201 Varick Street, Suite 1140
New York NY 10014

UNIVERSITY OF CALIFORNIA/LOS ALAMOS 1
NATIONAL LABORATORY
ATTN: DAN BACA/REPORT LIBRARIAN
P.O. BOX 1663, MS-P364
LOS ALAMOS NM 87545

RAND CORPORATION THE/LIBRARY 1
HELFER DORIS S/HEAD TECH SVCS
P.O. BOX 2138
SANTA MONICA CA 90406-2138

AEDC LIBRARY (TECH REPORTS FILE) 1
MS-10C
ARNOLD AFS TN 37389-9998

USAG 1
Attn: ASH-PCA-CRT
Ft Huachuca AZ 85613-6000

1839 EIG/EIET (KENNETH W. IREY) 1
KEESLER AFB MS 39534-6348

JTFPMC 1
Attn: Technical Director
1500 Planning Research Drive
McLean VA 22102

HQ ESC/CWPP 1
San Antonio TX 78243-5000

AFEWC/ESRI 4
SAN ANTONIO TX 78243-5000

485 EIG/EIER (DMC) 2
GRIFFISS AFB NY 13441-6348

ESD/AVS 1
ATTN: ADV SYS DEV
HANSCOM AFB MA 01731-5000

ESD/ICP 1
HANSCOM AFB MA 01731-5000

ESD/AVSE 2
BLDG 1704
HANSCOM AFB MA 01731-5000

HQ ESD SYS-2 1
HANSCOM AFB MA 01731-5000

ESD/TCO-2 1
ATTN: CAPTAIN J. MEYER
HANSCOM AFB MA 01731-5000

The Software Engineering Institute 1
Attn: Major Dan Burton, USAF
Jint Program Office
Carnegie Mellon University
Pittsburgh PA 15213-3890

DIRECTOR 1
NSA/CSS
ATTN: T513/TDL (DAVID MARJARUM)
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: W166
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: R24
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: R21
9800 SAVAGE ROAD
FORT GEORGE G MEASDE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: DEFSMAC (JAMES E. HILLMAN)
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR
NSA/CSS
ATTN: R31
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN: R5
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN: R8
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN: SC31
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN: S21
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN: V33 (S. Friedrich)
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN: W07
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN: W3
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR 2
NSA/CSS
ATTN: R523
FORT GEORGE G MEADE MD 20755-6000

DoD COMPUTER SECURITY CENTER 1
ATTN: C4/TIC
9800 SAVAGE ROAD
FORT GEORGE G MEADE MD 20755-6000

Bolt, Beranek, and Newman (BBN) 5
10 Moulton Street
Cambridge, MA 02238

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY (DARPA) 2
1400 Wilson Blvd
Arlington VA 22209